IBM® ILOG® CPLEX®

Concert Technology

version 12.1

C++ API

Reference Manual

2009

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Welcome to CPLEX Concert Technology

This reference manual documents the C++ API of IBM(R) ILOG(R) CPLEX(R) and Concert Technology.

Group Summary	
optim.concert	The IBM ILOG Concert API.
optim.concert.cplex	The IBM ILOG Concert CPLEX API.
optim.concert.extensions	The IBM ILOG Concert Extensions Library.
optim.concert.xml	The IBM ILOG Concert Serialization API.
optim.cplex.cpp	The API of CPLEX for users of C++.
optim.cplex.cpp.advanced	The advanced methods of the API of CPLEX for users of C++.

What Is CPLEX Concert Technology?

CPLEX Concert Technology offers a C++ library of classes and functions that enable you to design models of problems for both math programming (including linear programming, mixed integer programming, quadratic programming, and network programming) and constraint programming solutions.

This library is not a new programming language: it lets you use data structures and control structures provided by C++. Thus, the CPLEX Concert Technology part of an application can be completely integrated with the rest of that application (for example, the graphic interface, connections to databases, etc.) because it can share the same objects.

Furthermore, you can use the same objects to model your problem whether you choose a constraint programming or math programming approach. In fact, Concert Technology enables you to combine these technologies simultaneously.

What You Need to Know

This manual assumes that you are familiar with the operating system where you are using CPLEX Concert Technology. Since CPLEX Concert Technology is written for C++ developers, this manual assumes that you can write C++ code and that you have a working knowledge of your C++ development environment.

Notation

Throughout this manual, the following typographic conventions apply:

- Samples of code are written in this `typeface`.
- The names of constructors and member functions appear in this `typeface` in the section where they are documented.
- Important ideas are emphasized like *this*.

Naming Conventions

The names of types, classes, and functions defined in the CPLEX Concert Technology library begin with `Ilo`.

The names of classes are written as concatenated, capitalized words. For example:

IloNumVar

A lower case letter begins the first word in names of arguments, instances, and member functions. Other words in such a name begin with a capital letter. For example,

IloNumVarArray::setBounds

There are no public data members in CPLEX Concert Technology.

Accessors begin with the keyword `get` followed by the name of the data member. Accessors for Boolean members begin with `is` followed by the name of the data member. Like other member functions, the first word in such a name begins with a lower case letter, and any other words in the name begin with a capital letter.

Modifiers begin with the keyword `set` followed by the name of the data member.

Related Documents

The CPLEX Concert Technology library comes with the following documentation. The online documentation, in HTML format, may be accessed through standard HTML browsers.

- The Reference Manual documents the predefined C++ classes, global functions, type definitions, and macros in the libraries. It also provides formal explanations of certain Concepts, such as arrays, handles, notification, and column-wise modeling.
- The README file, delivered in the standard distribution, contains the most current information about platform prerequisites for CPLEX Concert Technology.
- Source code for examples is located in the examples directory in the standard distribution.

For More Information

CPLEX offers technical support, users' mailing lists, and comprehensive websites for the product, including Concert Technology.

Technical Support

For technical support of CPLEX Concert Technology, you should contact your local distributor, or, if you are a direct customer, contact the technical support center listed for your licensed product. We encourage you to use e-mail for faster, better service.

Web Sites

There are two kinds of web pages available to users of CPLEX Concert Technology: web pages restricted to owners of a paid maintenance contract; web pages freely available to all.

Web Pages for a Paid Maintenance Contract

The technical support pages on our world wide web sites contain FAQ (Frequently Asked/Answered Questions) and the latest patches for some of our products. Changes are posted in the product mailing list. Access to these pages is restricted to owners of an ongoing maintenance contract. The maintenance contract number and the name of the person this contract is sent to in your company will be needed for access, as explained on the login page.

All three of these sites contain the same information, but access is localized, so we recommend that you connect to the site corresponding to your location, and select the Services page from the home page.

- Americas: http://www.ilog.com
- Asia & Pacific Nations: http://www.ilog.com.sg
- Europe, Africa, and Middle East: http://www.ilog.fr

Web Pages for General Information

In addition to those web pages for technical support of a paid maintenance contract, you will find other web pages containing additional information about CPLEX Concert Technology, including technical papers that have also appeared at industrial and academic conferences, models developed by CPLEX and its customers, news about progress in optimization. This freely available information is located at these localized web sites:

- http://www.ilog.com/products/optimization/
- http://www.ilog.com.sg/products/optimization/
- http://www.ilog.fr/products/optimization/

Concepts

Arrays

For most basic classes (such as `IloNumVar` or `IloConstraint`) in Concert Technology, there is also a corresponding class of arrays where the elements of the array are instances of that basic class. For example, elements of an instance of `IloConstraintArray` are instances of the class `IloConstraint`.

Arrays in an Environment

Every array must belong to an environment (an instance of `IloEnv`). In other words, when you create a Concert Technology array, you pass an instance of `IloEnv` as a parameter to the constructor. All the elements of a given array must belong to the same environment.

Extensible Arrays

Concert Technology arrays are extensible. That is, you can add elements to the array dynamically. You add elements by means of the `add` member function of the array class.

You can also remove elements from an array by means of its `remove` member function.

References to elements of an array change whenever an element is added to or removed from the array. For example,

```
IloNumArray x;

IloNum *x1ptr = &(x[1]);
x.add(1.3);
*x1ptr no longer valid!
```

Arrays as Handles

Like other Concert Technology objects, arrays are implemented by means of two classes: a handle class corresponding to an implementation class. An object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of the corresponding implementation class. As a Concert Technology user, you will be working primarily with handles.

Copying Arrays

Many handles may point to the same implementation object. This principle holds true for arrays as well as other handle classes in Concert Technology. When you want to create more than one handle for the same implementation object, you should use either the copy constructor or the assignment operator of the array class. For example,

```
IloNumArray array(env);     // creates a handle pointing to new impl
IloNumArray array1(array);  // creates a handle pointing to same impl
IloNumArray array2;         // creates an empty handle
array2 = array;        // sets impl of handle array2 to impl of array
```

To take another example, the following lines add all elements of `a1` to `a2`, essentially copying the array.

```
IloNumArray a1;
IloNumArray a2;

a2.clear();
a2.add(a1);
```

Programming Hint: Using Arrays Efficiently

If your application only reads an array (that is, if your function does not modify an element of the array), then we recommend that you pass the array to your function as a `const` parameter. This practice forces Concert Technology to access the `const` conversion of the index operator (that is, `operator[]`), which is faster.

Assert and NDEBUG

Most member functions of classes in Concert Technology are inline functions that contain an `assert` statement. This statement asserts that the invoking object and the member function parameters are consistent; in some member functions, the `assert` statement checks that the handle pointer is non-null. These statements can be suppressed by the macro `NDEBUG`. This option usually reduces execution time. The price you pay for this choice is that attempts to access through null pointers are not trapped and usually result in memory faults.

Compilation with `assert` statements will not prevent core dumps by incorrect code. Instead, compilation with `assert` statements moves the execution of the incorrect code (the core dump, for example) to a place where you can see what is causing the problem in a source code debugger. Correctly written code will never cause one of these Concert Technology `assert` statements to fail.

Branch and cut

CPLEX uses *branch-and-cut search* when solving mixed integer programming (MIP) models. The branch-and-cut procedure manages a search tree consisting of *nodes*. Every node represents an LP or QP subproblem to be processed; that is, to be solved, to be checked for integrality, and perhaps to be analyzed further. Nodes are called *active* if they have not yet been processed. After a node has been processed, it is no longer active. Cplex processes active nodes in the tree until either no more active nodes are available or some limit has been reached.

A *branch* is the creation of two new nodes from a parent node. Typically, a branch occurs when the bounds on a single variable are modified, with the new bounds remaining in effect for that new node and for any of its descendants. For example, if a branch occurs on a binary variable, that is, one with a lower bound of 0 (zero) and an upper bound of 1 (one), then the result will be two new nodes, one node with a modified upper bound of 0 (the downward branch, in effect requiring this variable to take only the value 0), and the other node with a modified lower bound of 1 (the upward branch, placing the variable at 1). The two new nodes will thus have completely distinct solution domains.

A *cut* is a constraint added to the model. The purpose of adding any cut is to limit the size of the solution domain for the continuous LP or QP problems represented at the nodes, while not eliminating legal integer solutions. The outcome is thus to reduce the number of branches required to solve the MIP.

As an example of a cut, first consider the following constraint involving three binary (0-1) variables:

```
20x + 25y + 30z <= 40
```

That sample constraint can be strengthened by adding the following cut to the model:

```
1x + 1y + 1z <= 1
```

No feasible integer solutions are ruled out by the cut, but some fractional solutions, for example (0.0, 0.4, 1.0), can no longer be obtained in any LP or QP subproblems at the nodes, possibly reducing the amount of searching needed.

The branch-and-cut procedure, then, consists of performing branches and applying cuts at the nodes of the tree. Here is a more detailed outline of the steps involved.

First, the branch-and-cut tree is initialized to contain the root node as the only active node. The root node of the tree represents the entire problem, ignoring all of the explicit integrality requirements. Potential cuts are generated for the root node but, in the interest of keeping the problem size reasonable, not all such cuts are applied to the model immediately. If possible, an incumbent solution (that is, the best known solution that satisfies

all the integrality requirements) is established at this point for later use in the algorithm. Such a solution may be established either by CPLEX or by a user who specifies a starting solution by means of the Callable Library routine `CPXcopymipstart` or the Concert Technology method `IloCplex::setVectors`.

If you are solving a sequence of problems by modifying the problem already in memory and re-solving, then you do not need to establish a starting solution explicitly every time, because for each revised problem, the solution of the previous problem will be retained as a possible starting solution.

When processing a node, CPLEX starts by solving the continuous relaxation of its subproblem. that is, the subproblem without integrality constraints. If the solution violates any cuts, CPLEX may add some or all of them to the node problem and may resolve it, if CPLEX has added cuts. This procedure is iterated until no more violated cuts are detected (or deemed worth adding at this time) by the algorithm. If at any point in the addition of cuts the node becomes infeasible, the node is pruned (that is, it is removed from the tree).

Otherwise, CPLEX checks whether the solution of the node-problem satisfies the integrality constraints. If so, and if its objective value is better than that of the current incumbent, the solution of the node-problem is used as the new incumbent. If not, branching will occur, but first a heuristic method may be tried at this point to see if a new incumbent can be inferred from the LP-QP solution at this node, and other methods of analysis may be performed on this node. The branch, when it occurs, is performed on a variable where the value of the present solution violates its integrality requirement. This practice results in two new nodes being added to the tree for later processing.

Each node, after its relaxation is solved, possesses an optimal objective function value Z. At any given point in the algorithm, there is a node whose Z value is better (less, in the case of a minimization problem, or greater for a maximization problem) than all the others. This Best Node value can be compared to the objective function value of the incumbent solution. The resulting MIP Gap, expressed as a percentage of the incumbent solution, serves as a measure of progress toward finding and proving optimality. When active nodes no longer exist, then these two values will have converged toward each other, and the MIP Gap will thus be zero, signifying that optimality of the incumbent has been proven.

It is possible to tell CPLEX to terminate the branch-and-cut procedure sooner than a completed proof of optimality. For example, a user can set a time limit or a limit on the number of nodes to be processed. Indeed, with default settings, CPLEX will terminate the search when the MIP Gap has been brought lower than 0.0001 (0.01%), because it is often the case that much computation is invested in moving the Best Node value after the eventual optimal incumbent has been located. This termination criterion for the MIP Gap can be changed by the user, of course.

Callbacks in Concert Technology

A callback is an object with a `main` method implemented by a user. This user-written `main` method is called by the `IloCplex` algorithm at specific points during optimization.

Callbacks may be called repeatedly at various points during optimization; for each place a callback is called, CPLEX provides a separate callback class (derived from the class `IloCplex::CallbackI`). Such a callback class provides the specific API as a protected method to use for the particular implementation.

There are several varieties of callbacks:

- **Informational callbacks** allow your application to gather information about the progress of MIP optimization without interfering with performance of the search. In addition, an informational callback also enables your application to terminate optimization.
- **Query callbacks**, also known as diagnostic callbacks, enable your application to retrieve information about the progress of optimization, whether continuous or discrete. The information available depends on the algorithm (primal simplex, dual simplex, barrier, mixed integer, or network) that you are using. For example, a query callback can return the current objective value, the number of simplex iterations that have been completed, and other details. Query callbacks can also be called from presolve, probing, fractional cuts, and disjunctive cuts. Query callbacks may impede performance because the internal data structures that support query callbacks must be updated frequently. Furthermore, query or diagnostic callbacks make assumptions about the path of the search, assumptions that are correct with respect to conventional branch and cut but that may be false with respect to dynamic search. For this reason, query or diagnostic callbacks are **not** compatible with dynamic search. In other words, CPLEX

normally turns off dynamic search in the presence of query or diagnostic callbacks in an application.

- **Control callbacks** make it possible for you to define your own user-written routines and for your application to call those routines to interrupt and resume optimization. Control callbacks enable you to direct the search when you are solving a MIP in an instance of `IloCplex`. For example, control callbacks enable you to select the next node to process or to control the creation of subnodes (among other possibilities). Control callbacks are an advanced feature of CPLEX, and as such, they require a greater degree of familiarity with CPLEX algorithms. Because control callbacks can alter the search path in this way, control callbacks are **not** compatible with dynammic search. In other words, CPLEX normally turns off dynamic search in the presence of control callbacks in an application.

If you want to take advantage of dynamic search in your application, you should restrict your use of callbacks to the informational callbacks.

If you see a need for query, diagnostic, or control callbacks in your application, you can override the normal behavior of CPLEX by nondefault settings of the parameters `MIPSearch`, `ParallelMode`, and `Threads`. For more details about these parameters and their settings, see the *CPLEX Parameters Reference Manual*.

You do not create instances of the class `IloCplex::CallbackI`; rather, you use one of its child classes to implement your own callback. In order to implement your user-written callbacks with an instance of `IloCplex`, you should follow these steps:

1. 1. Determine which kind of callback you want to write, and choose the appropriate class for it. The class hierarchy (displayed online when you click Tree on the menu) may give you some ideas about which kind of callback suits your purpose.
2. 2. Derive your own subclass, `MyCallbackI`, say, from the appropriate predefined callback class.
3. 3. In your subclass of the callback class, use the protected API defined in the base class to implement the `main` routine of your user-written callback. (All constructors of predefined callback classes are protected; they can be called only from user-written derived subclasses.)
4. 4. In your subclass, implement the method `duplicateCallback`.
5. 5. Write a function `myCallback`, say, that creates an instance of your implementation class in the Concert Technology environment and returns it as an `IloCplex::Callback` handle.
6. 6. Create an instance of your callback class and pass it to the member function `IloCplex::use`.

There are macros of the form `ILOXXXCALLBACKn` (for n from 0 to 7) available to facilitate steps 2 through 5, where `XXX` stands for the particular callback under construction and n stands for the number of arguments that the function written in step 5 is to receive in addition to the environment argument.

You can use one instance of a callback with only one instance of `IloCplex`. When you use a callback with a second instance of `IloCplex`, a copy will be automatically created using the method `duplicateCallback`, and that copy will be used instead.

Also, an instance of `IloCplex` takes account of only one instance of a particular callback at any given time. That is, if you call `IloCplex::use` more than once with the same class of callback, the last call overrides any previous one. For example, you can use only one primal simplex callback at a time, or you can use only one network callback at a time; and so forth.

Existing extractables should never be modified within a callback. Temporary extractables, such as arrays, expressions, and range constraints, can be created and modified. Temporary extractables are often useful, for example, for computing cuts.

Example

Here is an example showing you how to terminate optimization after a given period of time if the solution is good enough. It uses one of the predefined macros to facilitate writing a control callback with a timer, a time limit, and a way to recognize a good enough solution.

```
ILOMIPINFOCALLBACK3(nodeLimitCallback,
                    IloBool, aborted,
                    IloNum,  nodeLimit,
                    IloNum,  acceptableGap)
{
   if ( !aborted  &&  hasIncumbent() ) {
      IloNum objval = getIncumbentObjValue();
      IloNum bound  = getBestObjValue();
```

```
      IloNum gap    = fabs(objval - bound) / (1.0 + fabs(bound)) * 100.0;
      if ( getNnodes() > nodeLimit &&
           gap < acceptableGap ) {
         getEnv().out() << endl
                        << "Good enough solution at "
                        << getNnodes() << " nodes, gap = "
                        << gap << "%, quitting." << endl;
         aborted = IloTrue;
         abort();
      }
   }
}
```

Column-Wise Modeling

Concert Technology supports column-wise modeling, a technique widely used in the math programming and operations research communities to build a model column by column. In Concert Technology, creating a new column is comparable to creating a new variable and adding it to a set of constraints. You use an instance of `IloNumColumn` to do so. An instance of `IloNumColumn` allows you to specify to which constraints or other extractable objects Concert Technology should add the new variable along with its data. For example, in a linear programming problem (an LP), if the new variable will appear in some linear constraints as ranges (instances of `IloRange`), you need to specify the list of such constraints along with the non zero coefficients (a value of `IloNum`) for each of them.

You then create a new column in your model when you create a new variable with an instance of `IloNumColumn` as its parameter. When you create the new variable, Concert Technology will add it along with appropriate parameters to all the extractable objects you have specified in the instance of `IloNumColumn`.

Instead of building an instance of `IloNumColumn`, as an alternative, you can use a column expression directly in the constructor of the variable. You can also use instances of `IloNumColumn` within column expressions.

The following undocumented classes provide the underlying mechanism for column-wise modeling:

- `IloAddValueToObj`
- `IloAddValueToRange`

The following operators are useful in column-wise modeling:

- in the class `IloRange`,

```
IloAddValueToRange operator() (IloNum value);
```

- in the class `IloObjective`,

```
IloAddValueToObj operator () (IloNum value);
```

That is, the `operator ()` in extractable classes, such as `IloRange` or `IloObjective`, creates descriptors of how Concert Technology should add the new, yet-to-be-created variable to the invoking extractable object.

You can use the `operator +` to link together the objects returned by `operator ()` to form a column. You can then use an instance of `IloNumColumn` to build up column expressions within a programming loop and thus save them for later use or to pass them to functions.

Here is how to use an instance of `IloNumColumn` with operators from `IloRange` and `IloObjective` to create a column with a coefficient of 2 in the objective, with 10 in `range1`, and with 3 in `range2`. The example then uses that column when it creates the new variable `newvar1`, and it uses column expressions when it creates `newvar2` and `newvar3`.

```
    IloNumColumn col = obj(2) + range1(10) + range2(3);
    IloNumVar newvar1(col);
    IloNumVar newvar2(col + range3(17));
    IloNumVar newvar3(range1(1) + range3(3));
```

In other words, given an instance `obj` of `IloObjective` and the instances `range1`, `range2`, and `range3` of `IloRange`, those lines create the new variables `newvar1`, `newvar2`, and `newvar3` and add them as linear terms to `obj`, `range1`, and `range3` in the following way:

```
obj: + 2*newvar1 + 2*newvar2
range1: +10*newvar1 + 10*newvar2 + 1*newvar3
range2: + 3*newvar1 + 3*newvar2
range3: + 17*newvar2 +3*newvar3
```

For more information, refer to the documentation of `IloNumColumn`,`IloObjective`, and `IloRange`.

Creation of Extractable Objects

For most Concert applications, you can simply create the extractable objects that you need to build your model, then let their destructors manage the subsequent deletions. However, when memory use is critical to your application, you may need to take control of the deletion of extractable objects. In such cases, you will need a deeper understanding of how Concert Technology creates and maintains extractable objects. The following guidelines, along with the concept Deletion of Extractable Objects, should help.

1. An **expression** (that is, an instance of the class `IloExpr`) is **passed by value** to an extractable object (an instance of the class `IloExtractable`). Therefore, you can delete the original expression after passing it by value without affecting the extractable object that received it.

Similarly, instances of `IloNumColumn` and `IloIntSet` are passed by value to any predefined Concert Technology objects. More generally, if you have multiple handles passed to Concert objects pointing to an instance of `IloExpr`, `IloNumColumn`, or `IloIntSet`, and you call a method that modifies one of those handles, Concert Technology performs a **lazy copy**. In other words, it first copies the implementation object for the handle you are modifying and then makes the modification. The other handles pointing to the original implementation object remain unchanged, and your modification has no impact on them.

Lazy copying does not apply to other Concert Technology objects. In general, it is recommended that you avoid using multiple handles to the same object if you do not feel comfortable with lazy copying.

2. A **variable** (that is, an instance of `IloNumVar`, `IloIntVar`, or `IloBoolVar`) is **passed by reference** to an extractable object. Therefore, when your Concert application is in linear deleter mode, deleting a variable will remove it from any extractables that received it.

3. An extractable object is **passed by reference** to a logical constraint (such as `IloIfThen`) or to a nonlinear expression (such as `IloMax`). Therefore, you should not delete the original expression after passing it to such functions unless you have finished with the associated model.

Here are some examples to consider in light of these guidelines. The first example illustrates guidelines 2 and 3.

```
IloEnv env;
IloNumVar x(env, 0, IloInfinity, "X");
IloNumVar y(env, 0, IloInfinity, "Y");
IloNumVar z(env, 0, IloInfinity, "Z");

IloExpr e = x + y;

IloConstraint c1 = (e <= z);
IloConstraint c2 = (e >= z);
IloConstraint c3 = IloIfThen(env, c1, c2);
e.end();           // OK; c1 and c2 use copies of e;
c1.end();          // BAD IDEA; c3 still references c1
IloModel m(env);
m.add (c3);        // c3 is not correctly represented in m.
```

In that example, since `c1` is passed by reference, the call to `c1.end` raises errors. In contrast, the call to `e.end` causes no harm because `e` is passed by value.

The following example illustrates guidelines 1 and 2.

```
      IloEnv env;
      IloModel model(env);
      IloNumVar y(env, 0, 10, "y");
#ifdef WILLDELETE
      IloNumVar y2 = y;     // second handle pointing to implementation of y
#else
      IloExpr y2 = y;       // first handle pointing to expression 1*y
#endif
      IloConstraint cst = y2 <= 6;
      model.add(cst);
      y2.end();
```

When `y2` is an instance of the class `IloNumVar`, the call to `y2.and` will remove `y2` from the constraint `cst`, according to guideline 2.

When `y2` is an expression, it will be passed by value to the constraint `cst`, according to guideline 1. Hence, the call to `y2.end` will leave `cst` untouched.

While a thorough understanding of these conventions provides you with complete control over management of the extractable objects in your application, in general, you should simply avoid creating extra handles to extractable objects that can result in unexpected behavior.

In light of that observation, the previous example can be simplified to the following lines:

```
    IloEnv env;
    IloModel model(env);
    IloNumVar y(env, 0, 10, "y");
    IloConstraint cst = y <= 6;
    model.add(cst);
```

Deletion of Extractable Objects

As a modeling layer, Concert allows the creation and destruction of extractables through the methods `IloExtractable::end` and `IloExtractableArray::endElements`. The purpose of these methods is to reclaim memory associated with the deleted objects while maintaining the safest possible Concert environment. In this context, a safe Concert environment is defined by the property that no object points to a deleted object; a pointer to a deleted object is referred to as a dangling pointer in C++.

There exist two paradigms to make sure of the safety of the delete operation. The first, linear mode, comes from math programming; in a Concert application, linear mode is possible only on extractable and other objects used in linear programming. The second, safe generic mode, is stricter and is valid on all Concert extractable objects.

You can access both paradigms by calling `IloEnv::setDeleter(IloDeleterMode mode)`, where mode may be `IloLinearDeleterMode` or `IloSafeDeleterMode`.

Linear Mode

To use linear mode, you must either

- call `IloEnv::setDeleter(IloLinearDeleterMode)`, or
- refrain from calling `IloEnv::setDeleter`, as linear is the default mode.

In linear mode, the following behavior is implemented:

- If a range constraint is deleted, it is removed from the models that contain it.
- If a variable is deleted, its coefficient is set to 0 (zero) in the ranges, expressions, and objectives where it appears. The variable is removed from the special ordered sets of type 1 and 2 (that is, SOS1 and SOS2), as well as from instances of `IloConversion` where it appears.

Example

This example tests the linear mode deletion of a variable `x`.

```
  void TestLinearDeleter() {
```

```
  IloEnv env;
  env.out() << "TestLinearDeleter" << endl;
  try {
    IloModel model(env);
    IloNumVar x(env, 0, 10, "x");
    IloNumVar y(env, 0, 10, "y");
    IloConstraint con = (x + y <= 0);
    IloConstraint con2 = y >= 6;
    IloNumVarArray ar(env, 2, x, y);
    IloSOS1 sos(env, ar, "sos");
    model.add(con);
    model.add(con2);
    model.add(sos);
    env.out() << "Before Delete" << endl;
    env.out() << model << endl;
    x.end();
    con2.end();
    env.out() << "After Delete" << endl;
    env.out() << model << endl;
  } catch (IloException& e) {
    cout << "Error : " << e << endl;
  }
  env.end();
}
```

The example produces the following output:

```
TestLinearDeleter
Before Delete
IloModel model0 = {
IloRange rng3(    1 * x + 1 * y ) <= 0

IloRange rng46 <=(    1 * y )

IloSOS1I (sos)
  _varArray [x(F)[0..10], y(F)[0..10]]
  _valArray []

}

After Delete
IloModel model0 = {
IloRange rng3(    1 * y ) <= 0

IloSOS1I (sos)
  _varArray [y(F)[0..10]]
  _valArray []
}
```

Safe Generic Mode

To use safe generic mode, you must:

- call `IloEnv::setDeleter(IloSafeDeleterMode)`, and
- add `#include <ilconcert/ilodeleter.h>` to your application.

In this mode, the environment builds a dependency graph between all extractable objects. This graph contains all extractable objects created

- after a call to `IloEnv::setDeleter(IloSafeDeleterMode)` and
- before a call to `IloEnv::unsetDeleter`.

Objects not managed by this dependency graph are referred to here as "nondeletable". An attempt to delete a nondeletable object will throw an exception.

It is recommended that you create this graph just after the creation of the environment and that you refrain from using `IloEnv::unsetDeleter` because building an incomplete dependency graph is very error prone and should only be attempted by advanced users. A clear example of this incomplete graph is the separation of a model between a nondeletable base model and deletable extensions of this base model.

Calling `IloExtractable::end` on extractable `xi` will succeed only if no other extractable object uses extractable `xi`. If this is not the case, a call to `IloExtractable::end` will throw an exception `IloDeleter::RequiresAnotherDeletionException` indicating which extractable object uses the extractable object that you want to delete.

Example

This example shows an attempt to delete one extractable object that is used by another.

```
void TestSafeDeleter() {
  IloEnv env;
  env.out() << "TestSafeDeleter" << endl;
  env.setDeleter(IloSafeDeleterMode);
  try {
    IloModel model(env);
    IloNumVar x(env, 0, 10);
    IloNumVar y(env, 0, 10);
    IloConstraint con = (x + y <= 0);
    try {
      x.end();
    } catch (IloDeleter::RequiresAnotherDeletionException &e) {
      cout << "Caught " << e << endl;
      e.getUsers()[0].end();
      e.end();
    }
    x.end();
  } catch (IloException& e) {
    cout << "Error : " << e << endl;
  }
  env.unsetDeleter();
  env.end();
}
```

The example produces the following output:

```
TestSafeDeleter
Caught You cannot end x1(F)[0..10] before IloRange rng3(  1 * x1 + 1 * x2  ) <= 0
```

To address this situation, you should use the method `IloExtractableArray::endElements`. With this method, all extractable objects in the array are deleted one after another. Thus, if an extractable object is used by another extractable object and this other extractable object is deleted before the first one, the system will not complain and will not throw an exception.

Example

This example illustrates the use of the `endElements` method

```
void TestSafeDeleterWithArray() {
  IloEnv env;
  env.out() << "TestSafeDeleterWithArray" << endl;
  env.setDeleter(IloSafeDeleterMode);
  try {
    IloModel model(env);
    IloNumVar x(env, 0, 10);
    IloNumVar y(env, 0, 10);
    IloConstraint con = (x + y <= 0);
    IloExtractableArray ar(env, 2, con, x);
    ar.endElements();
  } catch (IloException& e) {
    cout << "Error : " << e << endl;
  }
  env.unsetDeleter();
  env.end();
}
```

That example will not throw an exception.

Note

In this last example, the constraint `con` must appear before the variable `x` as it will be deleted before the variable `x`.

Exceptions, Errors

An exception is thrown; it is not allocated in a Concert Technology environment; it is not allocated on the C++ heap. It is not necessary for you as a programmer to delete an exception explicitly. Instead, the system calls the constructor of the exception to create it, and the system calls the destructor of the exception to delete it.

When exceptions are enabled on a platform that supports C++ exceptions, an instance of a class of Concert Technology is able to throw an exception in case of error. On platforms that do not support C++ exceptions, it is possible for Concert Technology to exit in case of error.

Programming Hint: Throwing and Catching Exceptions

Exceptions are thrown by value. They are not allocated on the C++ heap, nor in a Concert Technology environment. The correct way to catch an exception is to catch a reference to the error (indicated by the ampersand `&`), like this:

```
catch(IloException& oops);
```

Extraction

Concert Technology offers classes for you to design a *model* of your problem. You can then invoke an algorithm to extract information from your model to solve the problem. In this context, an algorithm is an instance of a class such as `IloCplex`, documented in the *CPLEX Reference Manual*, or `IloCP`, documented in the *CP Optimizer Reference Manual*.

For details about what each algorithm extracts from a model, see the reference manual documenting that algorithm. For example, the *CPLEX Reference Manual* lists precisely which classes of Concert Technology are extracted by an instance of `IloCplex`. In general terms, an instance of `IloCplex` extracts a model as rows and columns, where the columns indicate decision variables of the model. Also in general terms, an instance of `IloCP` extracts an instance of a class whose name begins `Ilo` to a corresponding instance of a class whose name begins `Ilc`. For example, an instance of `IloAllDiff` is extracted by `IloCP` as an instance of `IlcAllDiff`.

Goals

Goals can be used to control the branch and cut search in `IloCplex` . Goals are implemented in the class `IloCplex::GoalI` . The class `IloCplex::Goal` is the handle class. That is, it contains a pointer to an instance of `IloCplex::GoalI` along with accessors of objects in the implmenetation class.

To implement your own goal, you need to subclass `IloCplex::GoalI` and implement its virtual member functions `execute` and `duplicateGoal`. The method `execute` controls the branch-and-cut search. The method `duplicateGoal` creates a copy of the invoking goal object to be used for parallel branch-and-cut search. Implementing your goal can be greatly simplified if you use one of the macros `ILOCPLEXGOALn`.

Every branch-and-cut node maintains a goal stack, possibly containing `IloCplex::GoalI` objects. After `IloCplex` solves the relaxation of a node, it pops the top goal from the goal stack and calls its method `execute`. There are several types of goals:

- If `OrGoal` is executed, `IloCplex` will create child nodes. Each of the child nodes will be initialized with a copy of the goal stack of the current node. Then, for each child node, the specified goal in the `OrGoal` is pushed onto the corresponding goal stack of the child node. Finally, the current node is deleted. (See `IloCplex#GoalI::OrGoal` for a more detailed discussion.)

- If a cut goal is executed, the constraint will be added to the current node relaxation. Constraint goals are provided to represent both local and global cuts. Local cut goals are conventionally used to express branching.
- If `AndGoal` is executed, its subgoals are simply pushed onto the stack. (See `IloCplex::GoalI::AndGoal` for a more detailed discussion.)
- If `IloCplex::GoalI::FailGoal` is executed, `IloCplex` will prune the current node; that is, it will discontinue the search at the current node. `IloCplex` will continue with another node if there is one still available in the tree.
- If `IloCplex::GoalI::SolutionGoal` is executed, `IloCplex` will attempt to inject a user-provided solution as a new incumbent. Before CPLEX accepts the injected solution, it first tests whether the injected solution is feasible with respect to the model and goals.
- When CPLEX executes any other goal, the returned goal is simply pushed onto the stack.

`IloCplex` continues popping goals from the goal stack until `OrGoal` or `FailGoal` is executed, or until the stack becomes empty; in the case of an empty stack, it will continue with a built-in search strategy.

The predefined goals `OrGoal` and `AndGoal` allow you to combine goals. `AndGoal` allows you to execute different goals at one node, while `OrGoal` allows you to execute different goals on different, newly created nodes. A conventional use of these two goals in a return statement of a user-written goal looks like this:

```
return AndGoal ( OrGoal (branch1, branch2), this);
```

This `AndGoal` first pushes `this` (the goal currently being executed) onto the goal stack, and then it pushes the `OrGoal`. Thus the `OrGoal` is on top of the stack and will be executed next. When the `OrGoal` executes, it creates two new nodes and copies the remaining goal stack to both of them. Thus both new nodes will have `this` goal on top of the goal stack at this point. Then the `OrGoal` proceeds to push `branch1` onto the goal stack of the first child node and `branch2` onto the goal stack of the second goal child node. Conventionally, `branch1` and `branch2` contain cut goals, so by executing `branch1` and `branch2` at the respective child nodes, the child nodes will be restricted to represent smaller subproblems than their parent. After `branch1` and `branch2` have been executed, `this` is on top of the node stack of both child nodes; that is, both child nodes will continue branching according to the same rule. In summary, this example creates the branches `branch1` and `branch2` and continues in both branches to control the same search strategy `this`.

To perform a search using a goal, you need to solve the extracted model by calling the method `IloCplex::solve(goal)` with the goal to use as an argument instead of the standard `IloCplex::solve`. The method `solve(goal)` simply pushes the `goal` onto the goal stack of the root node before calling the standard `solve`.

See Also

`IloCplex::Goal` and `IloCplex::GoalI`

Handle Class

Most Concert Technology entities are implemented by means of two classes: a handle class and an implementation class, where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of the corresponding implementation class. As a Concert Technology user, you will be working primarily with handles.

As handles, these objects should be passed in either of these ways:

- as `const` by value (when no change is involved);
- by reference (when the function to which the handle is passed changes the implementation of that handle).

They should be created as automatic objects, where "automatic" has the usual C++ meaning.

Member functions of a handle class correspond to member functions of the same name in the implementation class.

Infeasibility Tools

When you problem is infeasible, CPLEX offers tools to help you diagnose the cause or causes of infeasibility in your model and possibly repair it: `IloCplex::refineConflict` and `IloCplex::feasOpt`.

Conflict Refiner

Given an infeasible model, the conflict refiner can identify conflicting constraints and bounds within the model to help you identify the causes of the infeasibility. In this context, a conflict is a subset of the constraints and bounds of the model which are mutually contradictory. The conflict refiner first examines the full infeasible model to identify portions of the conflict that it can remove. By this process of refinement, the conflict refiner arrives at a minimal conflict. A minimal conflict is usually smaller than the full infeasible model and thus makes infeasibility analysis easier. To invoke the conflict refiner, call the method `IloCplex::refineConflict`.

If a model happens to include multiple independent causes of infeasibility, then it may be necessary for the user to repair one such cause and then repeat the diagnosis with further conflict analysis.

A conflict does not provide information about the magnitude of change in data values needed to achieve feasibility. The techniques that CPLEX uses to refine a conflict include or remove constraints or bounds in trial conflicts; the techniques do not vary the data in constraints nor in bounds. To gain insight about changes in bounds on variables and constraints, consider the FeasOpt feature.

Also consider FeasOpt for an approach to automatic repair of infeasibility.

Refining a conflict in an infeasible model as defined here is similar to finding an irreducibly inconsistent set (IIS), an established technique in the published literature, long available within CPLEX. Both tools (conflict refiner and IIS finder) attempt to identify an infeasible subproblem in an infeasible model. However, the conflict refiner is more general than the IIS finder. The IIS finder is applicable only in continuous (that is, LP) models, whereas the conflict refiner can work on any type of problem, even mixed integer programs (MIP) and those containing quadratic elements (QP or QCP).

Also the conflict refiner differs from the IIS finder in that a user may organize constraints into one or more groups for a conflict. When a user specifies a group, the conflict refiner will make sure that either the group as a whole will be present in a conflict (that is, all its members will participate in the conflict, and removal of one will result in a feasible subproblem) or that the group will not participate in the conflict at all.

See the method `IloCplex::refineConflictExt` for more about groups.

A user may also assign a numeric preference to constraints or to groups of constraints. In the case of an infeasible model having more than one possible conflict, preferences guide the conflict refiner toward identifying constraints in a conflict as the user prefers.

In these respects, the conflict refiner represents an extension and generalization of the IIS finder.

FeasOpt

Alternatively, after a model have been proven infeasible, `IloCplex::feasOpt` performs an additional optimization that computes a minimal relaxation of the constraints over variables, of the bounds on variables, and of the righthand sides of constraints to make the model feasible. The parameter `FeasOptMode` lets you guide `feasOpt` in its computation of this relaxation.

`IloCplex::feasOpt` works in two phases. In its first phase, it attempts to minimize its relaxation of the infeasible model. That is, it attempts to find a feasible solution that requires minimal change. In its second phase, it finds an optimal solution among those that require only as much relaxation as it found necessary in the first phase.

Your choice of values for the parameter `FeasOptMode` indicates two aspects to CPLEX:

- whether to stop in phase one or continue to phase two:

    - Min means stop in phase one with a minimal relaxation.
    - Opt means continue to phase two for an optimum among those minimal relaxations.
- how to measure the minimality of the relaxation:
    - Sum means CPLEX should minimize the sum of all relaxations
    - Inf means that CPLEX should minimize the number of constraints and bounds relaxed.

The possible values of `FeasOptMode` are documented in the method `IloCplex::feasOpt`.

The status of the bounds and constraints of a relaxation returned by a call of `IloCplex::feasOpt` are documented in the enumeration `IloCplex::Status`.

Logical Constraints

For CPLEX, a logical constraint combines linear constraints by means of logical operators, such as logical and, logical or, negation (that is, not), conditional statements (that is, if ... then ...) to express complex relations between linear constraints. CPLEX can also handle certain logical expressions appearing within a linear constraint. One such logical expression is the minimum of a set of variables. Another such logical expression is the absolute value of a variable.

In C++ applications, the class `IloCplex` can extract modeling objects to solve a wide variety of MIPs and LPs. Under some conditions, a problem expressed in terms of logical constraints may be equivalent to a continuous LP, rather than a MIP. In such a case, there is no need for branching during the search for a solution. Whether a problem (or parts of a problem) represented by logical terms can be modeled and solved by LP depends on the shape of those logical terms. In this context, shape means convex or concave in the formal, mathematical sense.

For more about convexity, see that topic in the *CPLEX User's Manual*.

In fact, the class `IloCplex` can extract logical constraints as well as some logical expressions. The logical constraints that `IloCplex` can extract are these:

- `IloAnd` which can also be represented by the overloaded operator &&;
- `IloOr` which can also be represented by the overloaded operator ||;
- `IloDiff` which can also be represented by the overloaded operator !=;
- `IloNot`, negation, which can also be represented by the overloaded operator !;
- `IloIfThen`
- == (that is, the equivalence relation)
- != (that is, the exclusive-or relation)

For examples of logical constraints in CPLEX, see the *CPLEX User's Manual*.

Normalization: Reducing Linear Terms

Normalizing is sometimes known as *reducing the terms* of a linear expression.

Linear expressions consist of terms made up of constants and variables related by arithmetic operations; for example, x + 3y is a linear expression of two terms consisting of two variables. In some expressions, a given variable may appear in more than one term, for example, x + 3y +2x. Concert Technology has more than one way of dealing with linear expressions in this respect, and you control which way Concert Technology treats expressions from your application.

In one mode, Concert Technology analyzes linear expressions that your application passes it and attempts to reduce them so that a given variable appears in only one term in the linear expression. This is the default mode. You set this mode with the member function `IloEnv::setNormalizer(IloTrue)`.

In the other mode, Concert Technology assumes that no variable appears in more than one term in any of the linear expressions that your application passes to Concert Technology. We call this mode assume normalized linear expressions. You set this mode with the member function `IloEnv::setNormalizer(IloFalse)`.

In classes such as `IloExpr` or `IloRange`, there are member functions that check the setting of the member function `IloEnv::setNormalizer` in the environment and behave accordingly. The documentation of those member functions indicates how they behave with respect to normalization.

When you set `IloEnv::setNormalizer(IloFalse)`, those member functions assume that no variable appears in more than one term in a linear expression. This mode may save time during computation, but it entails the risk that a linear expression may contain one or more variables, each of which appears in one or more terms. Such a case may cause certain assertions in member functions of a class to fail if you do not compile with the flag `-DNDEBUG`.

By default, those member functions attempt to reduce expressions. This mode may require more time during preliminary computation, but it avoids of the possibility of a failed assertion in case of duplicates.

For more information, refer to the documentation of `IloEnv`,`IloExpr`, and `IloRange`.

Notification

You may modify the elements of a model in Concert Technology. For example, you may add or remove constraints, change the objective, add or remove columns, add or remove rows, and so forth.

In order to maintain consistency between a model and the algorithms that may use it, Concert Technology notifies algorithms about changes in the objects that the algorithms have extracted. In this manual, member functions that are part of this notification system are indicated like this:

> **Note**
>
> This member function notifies Concert Technology algorithms about this change of this invoking object.

Piecewise Linearity

Some problems are most naturally represented by constraints over functions that are not purely linear but consist of linear segments. Such functions are sometimes known as piecewise linear.

How to Define a Piecewise Linear Function

To define a piecewise linear function in CPLEX, you need these components:

- the variable of the piecewise linear function;
- the breakpoints of the piecewise linear function;
- the slope of each segment (that is, the rate of increase or decrease of the function between two breakpoints);
- the geometric coordinates of at least one point of the function.

In other words, for a piecewise linear function of n breakpoints, you need to know n+1 slopes. Typically, the breakpoints of a piecewise linear function are specified as an array of numeric values. The slopes of its segments are indicated as an array of numeric values as well. The geometric coordinates of at least one point of the function must also be specified. Then in CPLEX, those details are brought together by the global function `IloPiecewiseLinear`.

Another way to specify a piecewise linear function is to give the slope of the first segment, two arrays for the coordinates of the breakpoints, and the slope of the last segment.

For examples of these ways of defining a piecewise linear function, see this topic in the *CPLEX User's Manual*.

Discontinuous Piecewise Linear Functions

Intuitively, in a continuous piecewise linear function, the endpoint of one segment has the same coordinates as the initial point of the next segment. There are piecewise linear functions, however, where two consecutive breakpoints may have the same x coordinate but differ in the value of f(x). Such a difference is known as a *step* in the piecewise linear function, and such a function is known as *discontinuous*.

Syntactically, a step is represented in this way:

- The value of the first point of a step in the array of slopes is the height of the step.
- The value of the second point of the step in the array of slopes is the slope of the function after the step.

By convention, a breakpoint belongs to the segment that starts at that breakpoint.

In CPLEX, a discontinuous piecewise linear function is also represented as an instance of the class created by the global function `IloPiecewiseLinear`.

For examples of discontinuous piecewise linear functions, see this topic in the *CPLEX User's Manual*.

Using IloPiecewiseLinear

Whether it represents a continuous or a discontinuous piecewise linear function, an instance of the class created by the global function `IloPiecewiseLinear` behaves like a floating-point expression. That is, you may use it in a term of a linear expression or in a constraint added to a model (an instance of `IloModel`).

Unboundedness

The treatment of models that are unbounded involves a few subtleties. Specifically, a declaration of unboundedness means that CPLEX has determined that the model has an unbounded ray. Given any feasible solution x with objective z, a multiple of the unbounded ray can be added to x to give a feasible solution with objective z-1 (or z+1 for maximization models). Thus, if a feasible solution exists, then the optimal objective is unbounded. Note that CPLEX has not necessarily concluded that a feasible solution exists. Users can call the methods `IloCplex::isPrimalFeasible` and `IloCplex::isDualFeasible` to determine whether CPLEX has also concluded that the model has a feasible solution.

Group optim.concert

The IBM ILOG Concert API.

Class Summary	
IloAlgorithm	The base class of algorithms in Concert Technology.
IloAlgorithm::CannotExtractException	The class of exceptions thrown if an object cannot be extracted from a model.
IloAlgorithm::CannotRemoveException	The class of exceptions thrown if an object cannot be removed from a model.
IloAlgorithm::Exception	The base class of exceptions thrown by classes derived from IloAlgorithm.
IloAlgorithm::NotExtractedException	The class of exceptions thrown if an extractable object has no value in the current solution of an algorithm.
IloAnd	Defines a logical conjunctive-AND among other constraints.
IloArray	A template to create classes of arrays for elements of a given class.
IloBarrier	A system class to synchronize threads at a specified number.
IloBaseEnvMutex	A class to initialize multithreading in an application.
IloBoolArray	The array class of the basic Boolean class for a model.
IloBoolVar	An instance of this class represents a constrained Boolean variable in a Concert Technology model.
IloBoolVarArray	The array class of the Boolean variable class.
IloCondition	Provides synchronization primitives adapted to Concert Technology for use in a parallel application.
IloConstraint	An instance of this class is a constraint in a model.
IloConstraintArray	The array class of constraints for a model.
IloDiff	Constraint that enforces inequality.
IloEmptyHandleException	The class of exceptions thrown if an empty handle is passed.
IloEnv	The class of environments for models or algorithms in Concert Technology.
IloEnvironmentMismatch	This exception is thrown if you try to build an object using objects from another environment.
IloException	Base class of Concert Technology exceptions.
IloExpr	An instance of this class represents an expression in a model.
IloExprArray	The array class of the expressions class.
IloExpr::LinearIterator	An iterator over the linear part of an expression.
IloExtractable	Base class of all extractable objects.
IloExtractableArray	An array of extractable objects.
IloExtractableVisitor	The class for inspecting all nodes of an expression.
IloFastMutex	Synchronization primitives adapted to the needs of Concert Technology.
IloIfThen	This class represents a condition constraint.
IloIntArray	The array class of the basic integer class.
IloIntExpr	The class of integer expressions in Concert Technology.
IloIntExprArg	A class used internally in Concert Technology.

IloIntExprArray	The array class of the integer expressions class.
IloIntTupleSet	Ordered set of values represented by an array.
IloIntTupleSetIterator	Class of iterators to traverse enumerated values of a tuple-set.
IloIntVar	An instance of this class represents a constrained integer variable in a Concert Technology model.
IloIntVarArray	The array class of the integer constrained variables class.
IloIterator	A template to create iterators for a class of extractable objects.
IloModel	Class for models.
IloModel::Iterator	Nested class of iterators to traverse the extractable objects in a model.
IloMutexDeadlock	The class of exceptions thrown due to mutex deadlock.
IloMutexNotOwner	The class of exceptions thrown.
IloMutexProblem	Exception.
IloNot	Negation of its argument.
IloNumArray	The array class of the basic floating-point class.
IloNumExpr	The class of numeric expressions in a Concert model.
IloNumExprArg	A class used internally in Concert Technology.
IloNumExprArray	The array class of the numeric expressions class.
IloNumExpr::NonLinearExpression	The class of exceptions thrown if a numeric constant of a nonlinear expression is set or queried.
IloNumVar	An instance of this class represents a numeric variable in a model.
IloNumVarArray	The array class of IloNumVar.
IloObjective	An instance of this class is an objective in a model.
IloOr	Represents a disjunctive constraint.
IloRandom	This handle class produces streams of pseudo-random numbers.
IloRange	An instance of this class is a range in a model.
IloRangeArray	The array class of ranges for a model.
IloSemaphore	Provides synchronization primitives.
IloSolution	Instances of this class store solutions to problems.
IloSolutionIterator	This template class creates a typed iterator over solutions.
IloSolution::Iterator	It allows you to traverse the variables in a solution.
IloSolutionManip	An instance of this class accesses a specific part of a solution.
IloTimer	Represents a timer.

Typedef Summary	
IloBool	Type for Boolean values.
IloCplex::Status	An enumeration for the class `IloAlgorithm`.
IloInt	Type for signed integers.
IloNum	Type for numeric values as floating-point numbers.
IloSolutionArray	Type definition for arrays of `IloSolution` instances.

Macro Summary	
IloHalfPi	Half pi.
IloPi	Pi.

IloQuarterPi	Quarter pi.
ILOSTLBEGIN	Macro for STL.
IloThreeHalfPi	Three half-pi.
IloTwoPi	Two pi.

Enumeration Summary	
IloAlgorithm::Status	An enumeration for the class `IloAlgorithm`.
IloNumVar::Type	An enumeration for the class IloNumVar.
IloObjective::Sense	Specifies objective as minimization or maximization.

Function Summary	
IloAbs	Returns the absolute value of its argument.
IloAdd	Template to add elements to a model.
IloArcCos	Trigonometric functions.
IloCeil	Returns the least integer value not less than its argument.
IloDisableNANDetection	Disables NaN (Not a number) detection.
IloDiv	Integer division function.
IloEnableNANDetection	Enables NaN (Not a number) detection.
IloEndMT	Ends multithreading.
IloExponent	Returns the exponent of its argument.
IloFloor	Returns the largest integer value not greater than the argument.
IloGetClone	Creates a clone.
IloInitMT	Initializes multithreading.
IloIsNAN	Tests whether a double value is a NaN.
IloLexicographic	Returns a constraint which maintains two arrays to be lexicographically ordered.
IloLog	Returns the natural logarithm of its argument.
IloMax	Returns a numeric value representing the max of numeric values.
IloMaximize	Defines a maximization objective.
IloMin	Returns a numeric value representing the min of numeric values.
IloMinimize	Defines a minimization objective.
IloPiecewiseLinear	Represents a continuous or discontinuous piecewise linear function.
IloPower	Returns the power of its arguments.
IloRound	Computes the nearest integer value to its argument.
IloScalProd	Represents the scalar product.
IloScalProd	Represents the scalar product.
IloScalProd	Represents the scalar product.
IloScalProd	Represents the scalar product.
IloSquare	Returns the square of its argument.
IloSum	Returns a numeric value representing the sum of numeric values.
operator new	Overloaded C++ `new` operator.
operator!	Overloaded C++ operator for negation.
operator!=	Overloaded C++ operator.

operator%	Returns an expression equal to the modulo of its arguments.
operator%	Returns an expression equal to the modulo of its arguments.
operator&&	Overloaded C++ operator for conjunctive constraints.
operator*	Returns an expression equal to the product of its arguments.
operator+	Returns an expression equal to the sum of its arguments.
operator-	Returns an expression equal to the difference of its arguments.
operator/	Returns an expression equal to the quotient of its arguments.
operator<	overloaded C++ operator
operator<<	Overloaded C++ operator.
operator<<	Overloaded C++ operator.
operator<=	overloaded C++ operator
operator==	Overloaded C++ operator.
operator>	overloaded C++ operator
operator>=	overloaded C++ operator
operator>>	Overloaded C++ operator redirects input.
operator||	Overloaded C++ operator for a disjunctive constraint.

Variable Summary	
ILO_NO_MEMORY_MANAGER	OS environment variable controls Concert Technology memory manager.
IloInfinity	Largest double-precision floating-point number.
IloIntMax	Largest integer.
IloIntMin	Least integer.

Concert Technology offers a C++ library of classes and functions that enable you to design models of problems for both math programming (including linear programming, mixed integer programming, quadratic programming, and network programming) and constraint programming solutions.

Group optim.concert.cplex

The IBM ILOG Concert CPLEX API.

Class Summary	
IloConversion	For IBM ILOG CPLEX: a means to change the type of a numeric variable.
IloNumColumn	For IBM ILOG CPLEX: helps you design a model through column representation.
IloNumColumnArray	For IBM ILOG CPLEX: array class of the column representation class for a model.
IloSOS1	For IBM ILOG CPLEX: represents special ordered sets of type 1 (SOS1).
IloSOS1Array	For IBM ILOG CPLEX: the array class of special ordered sets of type 1 (SOS1).
IloSOS2	For IBM ILOG CPLEX: represents special ordered sets of type 2 (SOS2).
IloSOS2Array	For IBM ILOG CPLEX: the array class of special ordered sets of type 2 (SOS2).
IloSemiContVar	For IBM ILOG CPLEX: instance represents a constrained semicontinuous variable.
IloSemiContVarArray	For IBM ILOG CPLEX: is the array class of the semicontinuous numeric variable class for a model.

The IBM ILOG Concert API specific to CPLEX.

This group contains IBM ILOG Concert classes and functions specific to IBM ILOG CPLEX.

This group contains IBM ILOG Concert classes and functions specific to IBM ILOG CPLEX. (Other classes and functions of CPLEX are available in the group `optim.cplex.cpp`.)

Group optim.concert.extensions

The IBM ILOG Concert Extensions Library.

Class Summary	
IloCsvLine	Represents a line in a csv file.
IloCsvReader	Reads a formatted csv file.
IloCsvReader::IloColumnHeaderNotFoundException	Exception thrown for unfound header.
IloCsvReader::IloCsvReaderParameterException	Exception thrown for incorrect arguments in constructor.
IloCsvReader::IloDuplicatedTableException	Exception thrown for tables of same name in csv file.
IloCsvReader::IloFieldNotFoundException	Exception thrown for field not found.
IloCsvReader::IloFileNotFoundException	Exception thrown when file is not found.
IloCsvReader::IloIncorrectCsvReaderUseException	Exception thrown for call to inappropriate csv reader.
IloCsvReader::IloLineNotFoundException	Exception thrown for unfound line.
IloCsvReader::IloTableNotFoundException	Exception thrown for unfound table.
IloCsvReader::LineIterator	Line-iterator for csv readers.
IloCsvReader::TableIterator	Table-iterator of csv readers.
IloCsvTableReader	Reads a csv table with format.
IloCsvTableReader::LineIterator	Line-iterator for csv table readers.

Function Summary	
IloMax	Creates and returns a function equal to the maximal value of its argument functions.
IloMin	Creates and returns a function equal to the minimal value of its argument functions.
operator*	Creates and returns a function equal to its argument function multiplied by a given factor.
operator+	Creates and returns a function equal the sum of its argument functions.
operator-	Creates and returns a function equal to the difference between its argument functions.
operator<<	Overloaded operator for csv output.

Group optim.concert.xml

The IBM ILOG Concert Serialization API.

Class Summary
IloXmlContext
IloXmlInfo
IloXmlReader
IloXmlWriter

Group optim.cplex.cpp

The API of CPLEX for users of C++.

Class Summary
ControlCallbackI::PresolvedVariableException
IloBound
IloCplex
IloCplex::Aborter
IloCplex::BarrierCallbackI
IloCplex::BranchCallbackI
IloCplex::Callback
IloCplex::CallbackI
IloCplex::ContinuousCallbackI
IloCplex::ControlCallbackI
IloCplex::CrossoverCallbackI
IloCplex::CutCallbackI
IloCplex::DisjunctiveCutCallbackI
IloCplex::DisjunctiveCutInfoCallbackI
IloCplex::Exception
IloCplex::FlowMIRCutCallbackI
IloCplex::FlowMIRCutInfoCallbackI
IloCplex::FractionalCutCallbackI
IloCplex::FractionalCutInfoCallbackI
IloCplex::Goal
IloCplex::GoalI
IloCplex::HeuristicCallbackI
IloCplex::IncumbentCallbackI
IloCplex::InvalidCutException
IloCplex::LazyConstraintCallbackI
IloCplex::MIPCallbackI
IloCplex::MIPInfoCallbackI
IloCplex::MultipleConversionException
IloCplex::MultipleObjException
IloCplex::NetworkCallbackI
IloCplex::NodeCallbackI
IloCplex::NodeEvaluator
IloCplex::NodeEvaluatorI
IloCplex::OptimizationCallbackI
IloCplex::ParameterSet
IloCplex::PresolveCallbackI
IloCplex::ProbingCallbackI

IloCplex::ProbingInfoCallbackI
IloCplex::SearchLimit
IloCplex::SearchLimitI
IloCplex::SimplexCallbackI
IloCplex::SolveCallbackI
IloCplex::TuningCallbackI
IloCplex::UnknownExtractableException
IloCplex::UserCutCallbackI
MIPCallbackI::NodeData
ParameterSet::Iterator

Typedef Summary
ControlCallbackI::IntegerFeasibilityArray
GoalI::IntegerFeasibilityArray
IloCplex::BasisStatusArray
IloCplex::BranchDirectionArray
IloCplex::ConflictStatusArray

Macro Summary
ILOBARRIERCALLBACK0
ILOBRANCHCALLBACK0
ILOCONTINUOUSCALLBACK0
ILOCPLEXGOAL0
ILOCROSSOVERCALLBACK0
ILOCUTCALLBACK0
ILODISJUNCTIVECUTCALLBACK0
ILODISJUNCTIVECUTINFOCALLBACK0
ILOFLOWMIRCUTCALLBACK0
ILOFLOWMIRCUTINFOCALLBACK0
ILOFRACTIONALCUTCALLBACK0
ILOFRACTIONALCUTINFOCALLBACK0
ILOHEURISTICCALLBACK0
ILOINCUMBENTCALLBACK0
ILOLAZYCONSTRAINTCALLBACK0
ILOMIPCALLBACK0
ILOMIPINFOCALLBACK0
ILONETWORKCALLBACK0
ILONODECALLBACK0
ILOPRESOLVECALLBACK0
ILOPROBINGCALLBACK0
ILOPROBINGINFOCALLBACK0
ILOSIMPLEXCALLBACK0

ILOSOLVECALLBACK0
ILOTUNINGCALLBACK0
ILOUSERCUTCALLBACK0

Enumeration Summary
BranchCallbackI::BranchType
Callback::Type
ControlCallbackI::IntegerFeasibility
GoalI::BranchType
GoalI::IntegerFeasibility
IloBound::Type
IloCplex::Algorithm
IloCplex::BasisStatus
IloCplex::BoolParam
IloCplex::BranchDirection
IloCplex::ConflictStatus
IloCplex::CplexStatus
IloCplex::CutType
IloCplex::DeleteMode
IloCplex::DualPricing
IloCplex::IntParam
IloCplex::MIPEmphasisType
IloCplex::MIPStartEffort
IloCplex::MIPsearch
IloCplex::NodeSelect
IloCplex::NumParam
IloCplex::Parallel_Mode
IloCplex::PrimalPricing
IloCplex::Quality
IloCplex::Relaxation
IloCplex::StringParam
IloCplex::TuningStatus
IloCplex::VariableSelect
IloCplex::WriteLevelType

What is IloCplex?

`IloCplex` is a Concert Technology class derived from `IloAlgorithm`. Instances of this class are capable of solving optimization problems of the following types:

- Linear Programs (LPs),
- Mixed Integer Linear Programs (MILPs),
- Mixed Integer Programs (MIPs),
- Quadratic Programs (QPs),
- Mixed Integer Quadratic Programs (MIQPs),
- Quadratically Constrained Programs (QCPs);

- Mixed Integer Quadratically Constrained Programs (MIQCPs).

An instance of `IloCplex` can extract and solve models consisting of the following Concert Technology extractables:

Extractable Class	Used to Model
`IloNumVar`	numeric variables
`IloSemiContVar`	semi-continuous or semi-integer variables
`IloObjective`	at most one objective function with linear, piecewise linear, or quadratic expressions
`IloRange`	range constraints with linear or piecewise linear expressions
`IloConstraint`	ranged constraints of the form *expr1relationexpr*, where *expr1* indicates a linear, logical, or quadratic expression and the relation less than or equal to or the relation greater than or equal to; constraints can be combined by logical operators
`IloConversion`	variable type conversions
`IloModel`	submodels
`IloSOS1`	special ordered sets of type 1
`IloSOS2`	special ordered sets of type 2
`IloAnd`	constraint clauses

What is special about this set of extractable classes recognized by `IloCplex` is that models consisting of these objects can be transformed into mathematical programming problems of the form:

$$\begin{aligned} &\min/\max\ cx + \frac{1}{2}x^{T}Qx \\ &\text{s.t. } L \le Ax \le U \\ &l \le x \le u \end{aligned}$$

When all variables are continuous and Q is zero, problems of this form are known as Linear Programs (LPs). If Q is not zero, such problems are known as Quadratic Programs (QPs). If any variables are integer, semi-continuous, or Boolean, such problems are called Mixed Integer Programs (MIPs). A MIP with a zero Q matrix is called a Mixed Integer Linear Program (MILP), and a MIP with a non-zero Q is called a Mixed Integer Quadratic Program (MIQP). If there are quadratic constraints in the problem, and its variables are continuous, it is known as a Quadratically Constrained Program (QCP). If in addition to the quadratic constraints, there are discrete variables in the problem (such as integer, Boolean, or semi-continuous variables), then it is known as MIQCP.

- Objects of the class `IloNumVar` represent modeling variables. They are defined by the lower and upper bounds of the variable, and the type of the variable. The type of the variable can be one of these:
  - `ILOFLOAT`, for continuous,
  - `ILOINT`, for integer,
  - `ILOBOOL`, for Boolean variables.
- Objects of the class `IloSemiContVar` represent semi-continuous variables. A semi-continous variable may be 0 (zero) or may take a value within an interval defined by its semi-continuous lower and upper bounds. Semi-continuous variables are usually defined as continuous variables, but you can designate an instance of `IloSemiContVar` as integer by using the type indicator it inherits from `IloNumVar`.
- Objects of the class `IloObjective` represent objective functions of optimization models. `IloCplex` deals with models containing at most one objective function, and the objective function must be linear, piecewise linear, or quadratic.
- Objects of the class `IloRange` represent constraints of the form: `lower bound <= expression <= upper bound`. Any floating-point value or +/- `IloInfinity` can be used for the bounds.
- Objects of the class `IloConversion` change the type of a variable in a model. This class allows you to use the same variable with different types in different models.
- Objects of the class `IloModel` represent models which consist of extractable objects. They can be used to create submodels or additional models in a given environment.

- Objects of the class `IloSOS1` represent type 1 Special Ordered Sets (SOSs). A type 1 SOS specifies that at most one variable from a set of variables may take a nonzero value. Similarly, objects of the class `IloSOS2` represent type 2 SOSs. A type 2 SOS specifies that at most two variables from a set of variables may take nonzero values and that these two variables must be neighbors with respect to a specified order of the variables. SOS1 are rarely used and SOS2 are mostly used to model piecewise linear functions, for which Concert Technology provides direct support (with the class `IloPiecewiseLinear`).
- Objects of the class `IloAnd` are used in conjunction with objects of the class `IloSolution`.

IloCplex Optimizer Options

An instance of the class `IloCplex` is not really only one algorithm, but, in fact, consists of a set of highly configurable algorithms, also known as optimizer options. They include primal and dual simplex algorithms, barrier algorithm, a sifting algorithm, a network simplex algorithm, and a branch-and-cut algorithm for MIPs. Though in most cases `IloCplex` can be used like a black box, the optimizer options can be selected individually to provide a wealth of parameters that allow you to fine tune the algorithm to your particular model. In the case of the mixed integer optimizer, you can use your own goals or callbacks and directly control the branch-and-cut search carried out by `IloCplex`.

The most general kind of problem is a MIP. You might think of the LPs as a subset of MIPs: an LP is a problem in which the model is:

- without integer variables,
- without Boolean variables,
- without semi-continuous variables,
- without piecewise linear functions,
- without a quadratic component in the objective function,
- without quadratic constraints,
- and without a special ordered set (SOS).

For linear programming problems (LPs), a variety of additional solution information can be queried. These queries include dual information or, with the appropriate optimizer option, basis information. Sensitivity analysis allows you to analyze how you can modify your model while preserving the same solution. Or, if your model is infeasible, the infeasibility finder enables you to analyze the source of the infeasibility.

Group optim.cplex.cpp.advanced

The advanced methods of the API of CPLEX for users of C++.

Class Summary
IloCplex::BranchCallbackI
IloCplex::ControlCallbackI
IloCplex::CutCallbackI
IloCplex::Goal
IloCplex::GoalI
IloCplex::HeuristicCallbackI
IloCplex::IncumbentCallbackI
IloCplex::LazyConstraintCallbackI
IloCplex::NodeCallbackI
IloCplex::SolveCallbackI
IloCplex::UserCutCallbackI

These are advanced methods. Advanced methods typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

Class IloCplex::Aborter

Definition file: ilcplex/ilocplexi.h

An instance of this class gracefully terminates the solving and tuning methods of `IloCplex`. You can pass an instance of this class to one or more `IloCplex` objects. Calling the method `abort` will then terminate the solve or tuning method of the `IloCplex` object.

In particular, if you install an instance of this class in an instance of `IloCplex`, call the method `IloCplex::solve`, and later call the method `IloCplex::Aborter::abort`, then the `solve` will gracefully terminate, even if the methods are in separate threads. This convention makes it possible, for example, in a GUI application to terminate CPLEX when an end user presses a stop button.

Constructor Summary	
`public`	`Aborter(IloEnv env)`

Method Summary	
`public void`	`abort()`
`public void`	`clear()`
`public void`	`end()`
`public IloBool`	`isAborted() const`

Constructors

`public` **`Aborter`**`(IloEnv env)`

Constructs an instance of the Aborter class. It requires an instance of the same `IloEnv` as the `IloCplex` object with which to use the aborter.

Methods

`public void` **`abort`**`()`

Aborts the solving and tuning methods.

`public void` **`clear`**`()`

Clears the aborter.

`public void` **`end`**`()`

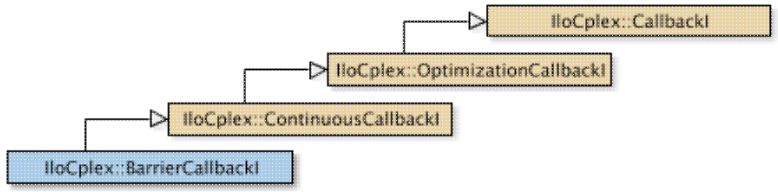
Ends the aborter.

`public IloBool` **`isAborted`**`() const`

Returns IIoTrue if `abort` has been called.

Class IloCplex::BarrierCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::BarrierCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a problem by means of the barrier optimizer. `IloCplex` calls the user-written callback after each iteration during optimization with the barrier method. If an attempt is made to access information not available to an instance of this class, an exception is thrown.

The constructor and methods of this class are for use in deriving a user-written callback class and in implementing the `main` method there.

For more information about the barrier optimizer, see the *CPLEX User's Manual*.

See Also: ILOBARRIERCALLBACK0, IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::ContinuousCallbackI, IloCplex::OptimizationCallbackI

Constructor Summary	
`protected`	`BarrierCallbackI(IloEnv env)`

Method Summary	
`public IloNum`	`getDualObjValue() const`

Inherited Methods from `ContinuousCallbackI`
`getDualInfeasibility, getInfeasibility, getNiterations, getObjValue, isDualFeasible, isFeasible`

Inherited Methods from `OptimizationCallbackI`
`getModel, getNcols, getNQCs, getNrows`

Inherited Methods from `CallbackI`
`abort, duplicateCallback, getEndTime, getEnv, main`

Constructors

`protected` **`BarrierCallbackI`**`(IloEnv env)`

This constructor creates a callback for use in an application of the barrier optimizer.

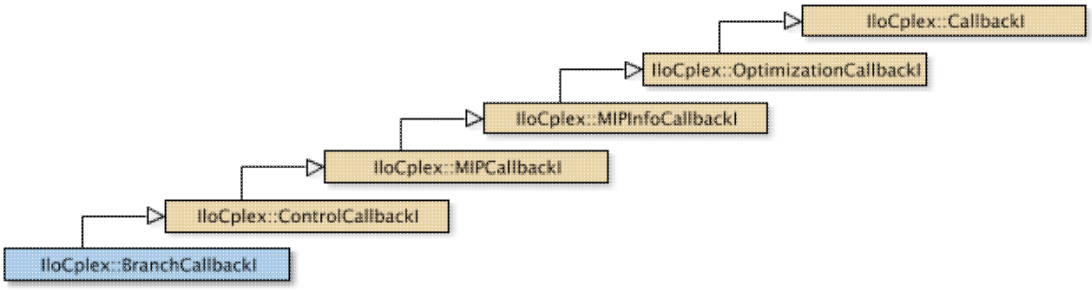
Methods

`public IloNum` **`getDualObjValue`**`() const`

This method returns the current dual objective value of the solution in the instance of `IloCplex` at the time the invoking callback is executed.

Class IloCplex::BranchCallbackI

Definition file: ilcplex/ilocplexi.h



Note

This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::BranchCallbackI` represents a user-written callback in an application that invokes an instance of `IloCplex` to solve a mixed integer program (MIP). The user-written callback is called prior to branching at a node in the branch-and-cut tree during the optimization of a MIP. It allows the user to query how the invoking instance of `IloCplex` is about to create subnodes at the current node and gives the user the option to override the selection made by the invoking instance of `IloCplex`. The user can create zero, one, or two branches.

- The method `BranchCallbackI::prune` removes the current node from the search tree. No subnodes from the current node will be added to the search tree.
- The method `BranchCallbackI::makeBranch` tells an instance of `IloCplex` how to create a subproblem. You may call this method zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; it you call it twice, it creates two nodes (one node at each call).
- If you call neither `IloCplex::BranchCallBackI::prune` nor `IloCplex::BranchCallBackI::makeBranch`, the instance of `IloCplex` proceeds with its own selection.
- Calling both `IloCplex::BranchCallBackI::prune` and `IloCplex::BranchCallBackI::makeBranch` in one invocation of a branch callback is an error and results in unspecified behavior.

The methods of this class are for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: ILOBRANCHCALLBACK0, IloCplex::BranchDirection, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI, IloCplex::ControlCallbackI, IloCplex::OptimizationCallbackI

Constructor Summary	
`protected`	`BranchCallbackI(IloEnv env)`

Method Summary	
`public IloNum`	`getBranch(IloNumVarArray vars, IloNumArray bounds, IloCplex::BranchDirectionArray dirs, IloInt i) const`

public BranchCallbackI::BranchType	getBranchType() const
public IloInt	getNbranches() const
public NodeId	getNodeId() const
public IloBool	isIntegerFeasible() const
public NodeId	makeBranch(const IloConstraintArray cons, const IloIntVarArray vars, const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)
public NodeId	makeBranch(const IloConstraintArray cons, const IloNumVarArray vars, const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)
public NodeId	makeBranch(const IloConstraint con, IloNum objestimate, NodeData * data=0)
public NodeId	makeBranch(const IloConstraintArray cons, IloNum objestimate, NodeData * data=0)
public NodeId	makeBranch(const IloIntVar var, IloNum bound, IloCplex::BranchDirection dir, IloNum objestimate, NodeData * data=0)
public NodeId	makeBranch(const IloNumVar var, IloNum bound, IloCplex::BranchDirection dir, IloNum objestimate, NodeData * data=0)
public NodeId	makeBranch(const IloIntVarArray vars, const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)
public NodeId	makeBranch(const IloNumVarArray vars, const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)
public void	prune()

Inherited Methods from `ControlCallbackI`
getDownPseudoCost, getDownPseudoCost, getFeasibilities, getFeasibilities, getFeasibility, getFeasibility, getFeasibility, getFeasibility, getLB, getLB, getLBs, getLBs, getNodeData, getObjValue, getSlack, getSlacks, getUB, getUB, getUBs, getUBs, getUpPseudoCost, getUpPseudoCost, getValue, getValue, getValue, getValues, getValues, isSOSFeasible, isSOSFeasible

Inherited Methods from `MIPCallbackI`
getNcliques, getNcovers, getNcuts, getNdisjunctiveCuts, getNflowCovers, getNflowPaths, getNfractionalCuts, getNGUBcovers, getNimpliedBounds, getNMIRs, getNzeroHalfCuts, getObjCoef, getObjCoef, getObjCoefs, getObjCoefs, getUserThreads

Inherited Methods from `MIPInfoCallbackI`
getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent

Inherited Methods from `OptimizationCallbackI`

getModel, getNcols, getNQCs, getNrows

Inherited Methods from `CallbackI`
abort, duplicateCallback, getEndTime, getEnv, main

Inner Enumeration
BranchCallbackI::BranchType

Constructors

```
protected BranchCallbackI(IloEnv env)
```

This constructor creates a branch callback, that is, a control callback for splitting a node into two branches.

Methods

```
public IloNum getBranch(IloNumVarArray vars, IloNumArray bounds,
IloCplex::BranchDirectionArray dirs, IloInt i) const
```

This method accesses branching information for the i-th branch that the invoking instance of `IloCplex` is about to create. The parameter `i` must be between 0 (zero) and `(getNbranches - 1)`; that is, it must be a valid index of a branch; normally, it will be zero or one.

A branch is normally defined by a set of variables and the bounds for these variables. Branches that are more complex cannot be queried. The return value is the node estimate for that branch.

- The parameter `vars` contains the variables for which new bounds will be set in the `i`-th branch.
- The parameter `bounds` contains the new bounds for the variables listed in `vars`; that is, `bounds[j]` is the new bound for `vars[j]`.
- The parameter `dirs` specifies the branching direction for the variables in `vars`.

```
dir[j] == IloCplex::BranchUp
```

means that `bounds[j]` specifies a lower bound for `vars[j]`.

```
dirs[j] == IloCplex::BranchDown
```

means that `bounds[j]` specifies an upper bound for `vars[j]`.

```
public BranchCallbackI::BranchType getBranchType() const
```

This method returns the type of branching `IloCplex` is going to do for the current node.

```
public IloInt getNbranches() const
```

This method returns the number of branches `IloCplex` is going to create at the current node.

```
public NodeId getNodeId() const
```

Returns the `NodeId` of the current node.

```
public IloBool isIntegerFeasible() const
```

This method returns `IloTrue` if the solution of the current node is integer feasible.

```
public NodeId makeBranch(const IloConstraintArray cons, const IloIntVarArray vars,
const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum
objestimate, NodeData * data=0)
```

This method offers the same facilities as the other methods `IloCplex::BranchCallbackI::makeBranch`, but for a branch specified by a set of constraints and a set of variables.

```
public NodeId makeBranch(const IloConstraintArray cons, const IloNumVarArray vars,
const IloNumArray bounds, const IloCplex::BranchDirectionArray dirs, IloNum
objestimate, NodeData * data=0)
```

This method offers the same facilities as the other methods `IloCplex::BranchCallbackI::makeBranch`, but for a branch specified by a set of constraints and a set of variables.

```
public NodeId makeBranch(const IloConstraint con, IloNum objestimate, NodeData *
data=0)
```

This method offers the same facilities for a branch specified by only one constraint as `IloCplex::BranchCallbackI::makeBranch` does for a branch specified by a set of constraints.

```
public NodeId makeBranch(const IloConstraintArray cons, IloNum objestimate,
NodeData * data=0)
```

This method overrides the branch chosen by an instance of `IloCplex`, by specifying a branch on constraints. A method named `makeBranch` can be called zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; it you call it twice, it creates two nodes (one node at each call). If you call it more than twice, it throws an exception.

- The parameter `cons` specifies an array of constraints that are to be added for the subnode being created.
- The parameter `objestimate` provides an estimate of the resulting optimal objective value for the subnode specified by this branch. The invoking instance of `IloCplex` may use this estimate to select nodes to process. Providing a wrong estimate will not influence the correctness of the solution, but it may influence performance. Using the objective value of the current node is usually a safe choice.
- The parameter `data` allows you to add an object of type `IloCplex::MIPCallbackI::NodeData` to the node representing the branch created by the `makeBranch` call. Such data objects must be instances of a user-written subclass of `IloCplex::MIPCallbackI::NodeData`.

```
public NodeId makeBranch(const IloIntVar var, IloNum bound,
IloCplex::BranchDirection dir, IloNum objestimate, NodeData * data=0)
```

For a branch specified by only one variable, this method offers the same facilities as `IloCplex::BranchCallbackI::makeBranch` for a branch specified by a set of variables.

```
public NodeId makeBranch(const IloNumVar var, IloNum bound,
```

```
IloCplex::BranchDirection dir, IloNum objestimate, NodeData * data=0)
```

For a branch specified by only one variable, this method offers the same facilities as `IloCplex::BranchCallbackI::makeBranch` for a branch specified by a set of variables.

```
public NodeId makeBranch(const IloIntVarArray vars, const IloNumArray bounds, const
IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)
```

This method overrides the branch chosen by an instance of `IloCplex`. A method named `makeBranch` can be called zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; it you call it twice, it creates two nodes (one node at each call). If you call it more than twice, it throws an exception.

Each call specifies a branch; in other words, it instructs the invoking `IloCplex` object how to create a subnode from the current node by specifying new, tighter bounds for a set of variables.

- The parameter `vars` contains the variables for which new bounds will be set in the branch.
- The parameter `bounds` contains the new bounds for the variables listed in `vars`; that is, `bounds[j]` is the new bound to be set for `vars[j]`.
- The parameter `dirs` specifies the branching direction for the variables in `vars`.

```
dir[j] == IloCplex::BranchUp
```

means that `bounds[j]` specifies a lower bound for `vars[j]`.

```
dirs[j] == IloCplex::BranchDown
```

means that `bounds[j]` specifies an upper bound for `vars[j]`.

- The parameter `objestimate` provides an estimate of the resulting optimal objective value for the subnode specified by this branch. The invoking instance of `IloCplex` may use this estimate to select nodes to process. Providing a wrong estimate will not influence the correctness of the solution, but it may influence performance. Using the objective value of the current node is usually a safe choice.
- The parameter `data` allows you to add an object of type `IloCplex::MIPCallbackI::NodeData` to the node representing the branch created by the `makeBranch` call. Such data objects must be instances of a user-written subclass of `IloCplex::MIPCallbackI::NodeData`.

```
public NodeId makeBranch(const IloNumVarArray vars, const IloNumArray bounds, const
IloCplex::BranchDirectionArray dirs, IloNum objestimate, NodeData * data=0)
```

This method overrides the branch chosen by an instance of `IloCplex`. A method named `makeBranch` can be called zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; it you call it twice, it creates two nodes (one node at each call). If you call it more than twice, it throws an exception.

Each call specifies a branch; in other words, it instructs the invoking `IloCplex` object how to create a subnode from the current node by specifying new, tighter bounds for a set of variables.

- The parameter `vars` contains the variables for which new bounds will be set in the branch.
- The parameter `bounds` contains the new bounds for the variables listed in `vars`; that is, `bounds[j]` is the new bound to be set for `vars[j]`.
- The parameter `dirs` specifies the branching direction for the variables in `vars`.

```
dir[j] == IloCplex::BranchUp
```

means that `bounds[j]` specifies a lower bound for `vars[j]`.

```
dirs[j] == IloCplex::BranchDown
```

means that `bounds[j]` specifies an upper bound for `vars[j]`.

- The parameter `objestimate` provides an estimate of the resulting optimal objective value for the subnode specified by this branch. The invoking instance of `IloCplex` may use this estimate to select nodes to process. Providing a wrong estimate will not influence the correctness of the solution, but it may influence performance. Using the objective value of the current node is usually a safe choice.
- The parameter `data` allows you to add an object of type `IloCplex::MIPCallbackI::NodeData` to the node representing the branch created by the `makeBranch` call. Such data objects must be instances of a user-written subclass of `IloCplex::MIPCallbackI::NodeData`.

```
public void prune()
```

By calling this method, you instruct the CPLEX branch-and-cut search not to create any child nodes from the current node, or, in other words, to discard nodes below the current node; it does not revisit the discarded nodes below the current node. In short, it creates no branches. It is an error to call both `prune` and `makeBranch` in one invocation of a callback.

Inner Enumerations

Enumeration BranchType

Definition file: ilcplex/ilocplexi.h

`IloCplex::BranchCallbackI::BranchType` is an enumeration limited in scope to the class `IloCplex::BranchCallbackI`. This enumeration is used by the method `IloCplex::BranchCallbackI::getBranchType` to tell what kind of branch `IloCplex` is about to do:

- `BranchOnVariable` specifies branching on a single variable.
- `BranchOnAny` specifies multiple bound changes and constraints will be used for branching.
- `BranchOnSOS1` specifies branching on an SOS of type 1.
- `BranchOnSOS2` specifies branching on an SOS of type 2.

See Also: IloCplex::BranchCallbackI

Fields:

`BranchOnVariable = CPX_TYPE_VAR`	= CPX_TYPE_VAR
`BranchOnSOS1 = CPX_TYPE_SOS1`	= CPX_TYPE_SOS1
`BranchOnSOS2 = CPX_TYPE_SOS2`	= CPX_TYPE_SOS2
`BranchOnAny = CPX_TYPE_ANY`	= CPX_TYPE_ANY
`UserBranch = CPX_TYPE_USER`	

Class IloCplex::Callback

Definition file: ilcplex/ilocplexi.h

This class is the handle class for all callback implementation classes available for `IloCplex`. Callback implementation classes are user-defined classes derived from a subclass of `IloCplex::CallbackI`.

See Also: IloCplex, IloCplex::CallbackI

Constructor Summary	
`public`	`Callback(IloCplex::CallbackI * impl=0)`

Method Summary	
`public void`	`end()`
`public IloCplex::CallbackI *`	`getImpl() const`
`public Callback::Type`	`getType() const`

Inner Enumeration
Callback::Type

Constructors

`public` **`Callback`**`(IloCplex::CallbackI * impl=0)`

This constructor creates a callback handle object and initializes it to the implementation object passed as the argument.

Methods

`public void` **`end`**`()`

This method deletes the implementation object pointed to by the invoking handle and sets the pointer to 0 (zero).

`public IloCplex::CallbackI *` **`getImpl`**`() const`

This method returns a pointer to the implementation object of the invoking handle.

`public Callback::Type` **`getType`**`() const`

This method returns the type of the callback implementation object referenced by the invoking handle.

Inner Enumerations

Enumeration Type

Definition file: ilcplex/ilocplexi.h

This enumeration type is used to identify the type of a callback implementation object referred to by an `IloCplex::Callback` handle.

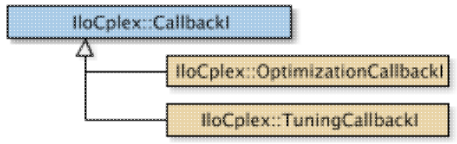
See Also: IloCplex::Callback

Fields:

```
Presolve = 0
Simplex = 1
Barrier = 2
Crossover = 3
Network = 4
MIP = 5
Probing = 6
FractionalCut = 7
DisjunctiveCut = 8
Branch = 9
Cut = 10
Node = 11
Heuristic = 12
Incumbent = 13
Solve = 14
FlowMIRCut = 15
Continuous = 16
MIPInfo = 17
ProbingInfo = 18
FractionalCutInfo = 19
DisjunctiveCutInfo = 20
FlowMIRCutInfo = 21
Tuning = 22
_Number = 23
```

Class IloCplex::CallbackI

Definition file: ilcplex/ilocplexi.h



This is the abstract base class for user-written callback classes. It provides their common application programming interface (API). Callbacks may be called repeatedly at various points during an optimization; for each place a callback is called, CPLEX provides a separate callback class (derived from this class). Such a callback class provides the specific API as a protected method to use for the particular implementation.

You do not create instances of this class; rather, you use one of its child classes to implement your own callback. In order to implement your user-written callbacks with an instance of `IloCplex`, you should follow these steps:

1. Determine which kind of callback you want to write, and choose the appropriate class for it. The class hierarchy in Tree may give you some ideas here.
2. Derive your own subclass, `MyCallbackI`, say, from the appropriate predefined callback class.
3. In your subclass of the callback class, use the protected API defined in the base class to implement the `main` routine of your user-written callback. (All constructors of predefined callback classes are protected; they can be called only from user-written derived subclasses.)
4. In your subclass, implement the method `duplicateCallback`.
5. Write a function `myCallback`, say, that creates an instance of your implementation class in the Concert Technology environment and returns it as a handle of `IloCplex::Callback`.
6. Create an instance of your callback class and pass it to the member function `IloCplex::use`.

Note

Macros `ILOXXXCALLBACKn` (for `n` from 0 to 7) are available to facilitate steps 2 through 5, where `XXX` stands for the particular callback under construction and `n` stands for the number of parameters that the function written in step 5 is to receive in addition to the environment parameter.

You can use one instance of a callback with only one instance of `IloCplex`. When you use a callback with a second instance of `IloCplex`, a copy will be automatically created using the method `duplicateCallback`, and that copy will be used instead.

Also, an instance of `IloCplex` takes account of only one instance of a particular callback at any given time. That is, if you call `IloCplex::use` more than once with the same class of callback, the last call overrides any previous one. For example, you can use only one primal simplex callback at a time, or you can use only one network callback at a time; and so forth.

There are two varieties of callbacks:

- Query callbacks enable your application to retrieve information about the current solution in an instance of `IloCplex`. The information available depends on the algorithm (primal simplex, dual simplex, barrier, mixed integer, or network) that you are using. For example, a query callback can return the current objective value, the number of simplex iterations that have been completed, and other details. Query callbacks can also be called from presolve, probing, fractional cuts, and disjunctive cuts.
- Control callbacks enable you to direct the search when you are solving a MIP in an instance of `IloCplex`. For example, control callbacks enable you to select the next node to process or to control the creation of subnodes (among other possibilities).

Existing extractables should never be modified within a callback. Temporary extractables, such as arrays, expressions, and range constraints, can be created and modified. Temporary extractables are often useful, for example, for computing cuts.

See Also: ILOBARRIERCALLBACK0, ILOBRANCHCALLBACK0, IloCplex, IloCplex::BarrierCallbackI, IloCplex::BranchCallbackI, IloCplex::Callback, IloCplex::ControlCallbackI, IloCplex::CrossoverCallbackI, IloCplex::CutCallbackI, IloCplex::DisjunctiveCutCallbackI, IloCplex::FlowMIRCutCallbackI, IloCplex::FractionalCutCallbackI, IloCplex::HeuristicCallbackI, IloCplex::IncumbentCallbackI, IloCplex::ContinuousCallbackI, IloCplex::MIPCallbackI, IloCplex::NetworkCallbackI, IloCplex::NodeCallbackI, IloCplex::OptimizationCallbackI, IloCplex::PresolveCallbackI, IloCplex::ProbingCallbackI, IloCplex::SimplexCallbackI, IloCplex::SolveCallbackI, IloCplex::TuningCallbackI, ILOCROSSOVERCALLBACK0, ILOCUTCALLBACK0, ILOBRANCHCALLBACK0, ILODISJUNCTIVECUTCALLBACK0, ILOFLOWMIRCUTCALLBACK0, ILOFRACTIONALCUTCALLBACK0, ILOHEURISTICCALLBACK0, ILOINCUMBENTCALLBACK0, ILOCONTINUOUSCALLBACK0, ILOMIPCALLBACK0, ILONETWORKCALLBACK0, ILONODECALLBACK0, ILOPRESOLVECALLBACK0, ILOPROBINGCALLBACK0, ILOSIMPLEXCALLBACK0, ILOSOLVECALLBACK0, ILOTUNINGCALLBACK0

Method Summary	
public void	abort()
protected virtual CallbackI *	duplicateCallback() const
public IloNum	getEndTime() const
public IloEnv	getEnv() const
protected virtual void	main()

Methods

`public void` **`abort`**`()`

This method instructs CPLEX to stop the current optimization after the user callback finishes. Note that executing additional `IloCplex` callback methods in the callback can lead to unpredictable behavior. For example, callback methods such as `IloCplex::SolveCallbackI::solve` or `IloCplex::BranchCallbackI::makeBranch` can overwrite the callback status and thus enable the optimization to continue. Therefore, to abort an optimization effectively, a user should exit the callback by one of the following ways:

- Call `return` immediately after the call of `abort`.
- Structure the callback so that it calls no additional methods of `IloCplex::CallbackI` and its subclasses after `abort`.

`protected virtual CallbackI *` **`duplicateCallback`**`() const`

This virtual method must be implemented to create a copy of the invoking callback object on the same environment. Typically the following implementation will work for a callback class called `MyCallbackI`:

```
IloCplex::CallbackI* MyCallbackI::duplicateCallback() const {
  return (new (getEnv()) MyCallbackI(*this));
}
```

This method is called by an `IloCplex` object in two cases:

- When `cplex.use(cb)` is called for a callback object `cb` that is already used by another instance of `IloCplex`, a copy of the implementation object of `cb` is created by calling `duplicateCallback` and used in its place. The method `use` will return a handle to that copy.
- When a parallel optimizer is used, `IloCplex` creates copies of every callback that it uses by calling `duplicateCallback`. One copy of a callback is created for each additional thread being used in the parallel optimizer. During the optimization, each thread will invoke the copy corresponding to the thread number. The methods provided by the callback APIs are guaranteed to be threadsafe. However, when accessing parameters passed to callbacks or members stored in a callback, it is up to the user to make

sure of thread safety by synchronizing access or creating distinct copies of the data in the implementation of `duplicateCallback`.

```
public IloNum getEndTime() const
```

This method returns a time stamp specifying when the time limit will occur. To compute remaining time in seconds, substract the result of `IloCplex::getCplexTime` from the result of this method. This computation yields either wall clock time (also known as real time) or CPU time, depending on the clock type set by the parameter `ClockType`. The absolute value of the time stamp is not meaningful.

```
public IloEnv getEnv() const
```

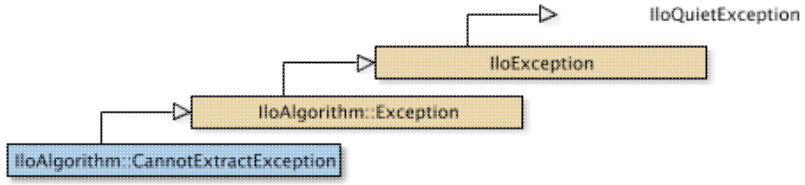
This method returns the environment belonging to the `IloCplex` object that invoked the method `main`.

```
protected virtual void main()
```

This virtual method is to be implemented by the user in a derived callback class to define the functionality of the callback. When an instance of `IloCplex` uses a callback (an instance of `IloCplex::CallbackI` or one of its derived subclasses), `IloCplex` calls this virtual method `main` at the point during the optimization at which the callback is executed.

Class IloAlgorithm::CannotExtractException

Definition file: ilconcert/iloalg.h



The class of exceptions thrown if an object cannot be extracted from a model.
If an attempt to extract an object from a model fails, this exception is thrown.

Method Summary	
public void	end()
public const IloAlgorithmI *	getAlgorithm() const
public IloExtractableArray &	getExtractables()

Inherited Methods from `IloException`
end, getMessage

Methods

```
public void end()
```

This member function deletes the invoking exception. That is, it frees memory associated with the invoking exception.

```
public const IloAlgorithmI * getAlgorithm() const
```

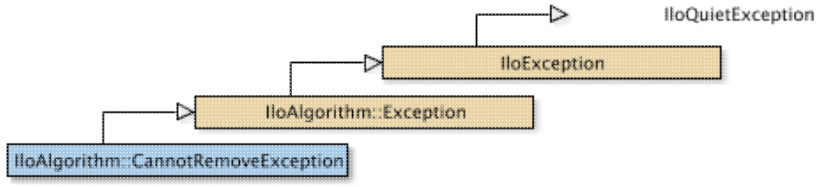
The member function **getAlgorithm** returns the algorithm from which the exception was thrown.

```
public IloExtractableArray & getExtractables()
```

The member function **getExtractables** returns the extractable objects that triggered the exception.

Class IloAlgorithm::CannotRemoveException

Definition file: ilconcert/iloalg.h



The class of exceptions thrown if an object cannot be removed from a model.
If an attempt to remove an extractable object from a model fails, this exception is thrown.

Method Summary	
public void	end()
public const IloAlgorithmI *	getAlgorithm() const
public IloExtractableArray &	getExtractables()

Inherited Methods from `IloException`
end, getMessage

Methods

`public void` **`end`**`()`

This member function deletes the invoking exception. That is, it frees memory associated with the invoking exception.

`public const IloAlgorithmI * ` **`getAlgorithm`**`() const`

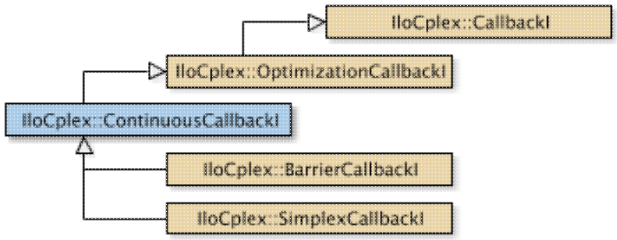
The member function **getAlgorithm** returns the algorithm from which the exception was thrown.

`public IloExtractableArray & ` **`getExtractables`**`()`

The member function **getExtractables** returns the extractable objects that triggered the exception.

Class IloCplex::ContinuousCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of a class derived from `IloCplex::ContinuousCallbackI` represents a user-written callback in a CPLEX application that uses an instance of `IloCplex` with the primal simplex, dual simplex, or barrier optimizer. `IloCplex` calls the user-written callback after each iteration during an optimization of a problem solved at a node. This class offers methods for use within the callbacks you write. In particular, there are methods in this class to access primal and dual feasibility, number of iterations, and objective value.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

Note

There are special callbacks for simplex and barrier, that is, `IloCplex::SimplexCallbackI` and `IloCplex::BarrierCallbackI`, respectively. Using a continous callback sets this callback in both of these algorithms. If a special callback was already set for one of these algorithms, (for example, simplex) it is replaced by the general continuous callback.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::OptimizationCallbackI, ILOCONTINUOUSCALLBACK0

Constructor Summary	
protected	ContinuousCallbackI(IloEnv env)

Method Summary	
public IloNum	getDualInfeasibility() const
public IloNum	getInfeasibility() const
public IloInt	getNiterations() const
public IloNum	getObjValue() const
public IloBool	isDualFeasible() const
public IloBool	isFeasible() const

Inherited Methods from `OptimizationCallbackI`
getModel, getNcols, getNQCs, getNrows

Inherited Methods from `CallbackI`
abort, duplicateCallback, getEndTime, getEnv, main

Constructors

`protected` **`ContinuousCallbackI`**`(IloEnv env)`

This constructor creates a callback for use in an application that solves continuous models.

Methods

`public IloNum` **`getDualInfeasibility`**`() const`

This method returns the current dual infeasibility measure of the solution in the instance of `IloCplex` at the time the invoking callback is executed.

`public IloNum` **`getInfeasibility`**`() const`

This method returns the current primal infeasibility measure of the solution in the instance of `IloCplex` at the time the invoking callback is executed.

`public IloInt` **`getNiterations`**`() const`

This method returns the number of iterations completed so far by an instance of `IloCplex` at the invoking callback is executed.

`public IloNum` **`getObjValue`**`() const`

This method returns the current objective value of the solution in the instance of `IloCplex` at the time the invoking callback is executed.

If you need the object representing the objective itself, consider the method `IloCplex::getObjective` instead.

`public IloBool` **`isDualFeasible`**`() const`

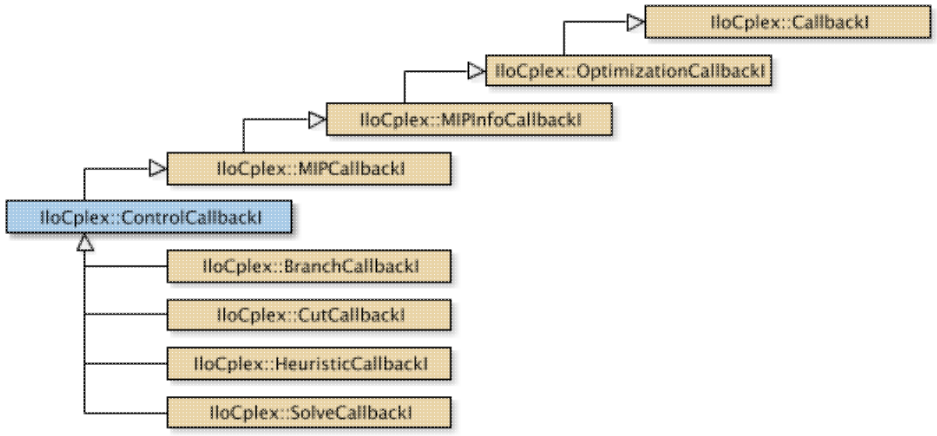
This method returns `IloTrue` if the current solution is dual feasible.

`public IloBool` **`isFeasible`**`() const`

This method returns `IloTrue` if the current solution is primal feasible.

Class IloCplex::ControlCallbackI

Definition file: ilcplex/ilocplexi.h



Note

This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

This class defines the common application programming interface (API) for the following classes that allow you to control the MIP search:

- `IloCplex::SolveCallbackI`
- `IloCplex::CutCallbackI`
- `IloCplex::HeuristicCallbackI`
- `IloCplex::BranchCallbackI`

An instance of one of these classes represents a user-written callback that intervenes in the search for a solution at a given node in an application that uses an instance of `IloCplex` to solve a mixed integer program (MIP). Control callbacks are tied to a node. They are called at each node during `IloCplex` branch-and-cut search. The user never subclasses the `IloCplex::ControlCallbackI` class directly; it only defines the common interface of thosee listed callbacks.

In particular, `SolveCallbackI` is called before solving the node relaxation and optionally allows substitution of its solution. `IloCplex` does this by default. After the node relaxation has been solved, either by an instance of `SolveCallbackI` or by `IloCplex`, the other control callbacks are called in the following order:

1. `IloCplex::CutCallbackI`
2. `IloCplex::HeuristicCallbackI`
3. `IloCplex::BranchCallbackI`

If the cut callback added new cuts to the node relaxation, the node relaxation will be solved again using the solve callback, if used. The same is true if `IloCplex` generated its own cuts.

The methods of this class are protected and its constructor is private; you cannot directly subclass this class; you must derive from its subclasses.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, ControlCallbackI::IntegerFeasibility, ControlCallbackI::IntegerFeasibilityArray, IloCplex::MIPCallbackI, IloCplex::OptimizationCallbackI

Method Summary	
public IloNum	getDownPseudoCost(const IloIntVar var) const
public IloNum	getDownPseudoCost(const IloNumVar var) const
public void	getFeasibilities(ControlCallbackI::IntegerFeasibilit stat, const IloIntVarArray var) const
public void	getFeasibilities(ControlCallbackI::IntegerFeasibilit stat, const IloNumVarArray var) const
public ControlCallbackI::IntegerFeasibility	getFeasibility(const IloIntVar var) const
public ControlCallbackI::IntegerFeasibility	getFeasibility(const IloNumVar var) const
public ControlCallbackI::IntegerFeasibility	getFeasibility(const IloSOS2 sos) const
public ControlCallbackI::IntegerFeasibility	getFeasibility(const IloSOS1 sos) const
public IloNum	getLB(const IloIntVar var) const
public IloNum	getLB(const IloNumVar var) const
public void	getLBs(IloNumArray val, const IloIntVarArray vars) const
public void	getLBs(IloNumArray val, const IloNumVarArray vars) const
public NodeData *	getNodeData() const
public IloNum	getObjValue() const
public IloNum	getSlack(const IloRange rng) const
public void	getSlacks(IloNumArray val, const IloRangeArray con) const
public IloNum	getUB(const IloIntVar var) const
public IloNum	getUB(const IloNumVar var) const
public void	getUBs(IloNumArray val, const IloIntVarArray vars) const
public void	getUBs(IloNumArray val, const IloNumVarArray vars) const
public IloNum	getUpPseudoCost(const IloIntVar var) const
public IloNum	getUpPseudoCost(const IloNumVar var) const
public IloNum	getValue(const IloIntVar var) const
public IloNum	getValue(const IloNumVar var) const
public IloNum	getValue(const IloExprArg expr) const
public void	getValues(IloNumArray val, const IloIntVarArray vars) const
public void	getValues(IloNumArray val, const IloNumVarArray vars) const
public IloBool	isSOSFeasible(const IloSOS2 sos2) const
public IloBool	isSOSFeasible(const IloSOS1 sos1) const

Inherited Methods from `MIPCallbackI`

getNcliques, getNcovers, getNcuts, getNdisjunctiveCuts, getNflowCovers, getNflowPaths, getNfractionalCuts, getNGUBcovers, getNimpliedBounds, getNMIRs, getNzeroHalfCuts, getObjCoef, getObjCoef, getObjCoefs, getObjCoefs, getUserThreads

Inherited Methods from `MIPInfoCallbackI`
getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent

Inherited Methods from `OptimizationCallbackI`
getModel, getNcols, getNQCs, getNrows

Inherited Methods from `CallbackI`
abort, duplicateCallback, getEndTime, getEnv, main

Inner Enumeration
ControlCallbackI::IntegerFeasibility

Inner Typedef
ControlCallbackI::IntegerFeasibilityArray

Inner Class
ControlCallbackI::PresolvedVariableException

Methods

```
public IloNum getDownPseudoCost(const IloIntVar var) const
```

This method returns the current pseudo cost for branching downward on the variable `var`.

```
public IloNum getDownPseudoCost(const IloNumVar var) const
```

This method returns the current pseudo cost for branching downward on the variable `var`.

```
public void getFeasibilities(ControlCallbackI::IntegerFeasibilityArray stat, const
IloIntVarArray var) const
```

This method specifies whether each of the variables in the array `vars` is integer feasible, integer infeasible, or implied integer feasible by putting the status in the corresponding element of the array `stats`.

```
public void getFeasibilities(ControlCallbackI::IntegerFeasibilityArray stat, const
IloNumVarArray var) const
```

This method specifies whether each of the variables in the array `vars` is integer feasible, integer infeasible, or implied integer feasible by putting the status in the corresponding element of the array `stats`.

```
public ControlCallbackI::IntegerFeasibility getFeasibility(const IloIntVar var)
const
```

This method specifies whether the variable `var` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public ControlCallbackI::IntegerFeasibility getFeasibility(const IloNumVar var)
const
```

This method specifies whether the variable `var` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public ControlCallbackI::IntegerFeasibility getFeasibility(const IloSOS2 sos) const
```

This method specifies whether the Special Ordered Set `sos` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public ControlCallbackI::IntegerFeasibility getFeasibility(const IloSOS1 sos) const
```

This method specifies whether the Special Ordered Set `sos` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public IloNum getLB(const IloIntVar var) const
```

This method returns the lower bound of `var` at the current node. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution value from `getValue` may violate this bound at a node where a new incumbent has been found because the bound is tightened when an incumbent is found.

Unbounded Variables

If a variable lacks a lower bound, then `getLB` returns a value greater than or equal to `-IloInfinity` for greater than or equal to constraints with no lower bound.

```
public IloNum getLB(const IloNumVar var) const
```

This method returns the lower bound of `var` at the current node. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution value from `getValue` may violate this bound at a node where a new incumbent has been found because the bound is tightened when an incumbent is found.

Unbounded Variables

If a variable lacks a lower bound, then `getLB` returns a value greater than or equal to `-IloInfinity` for greater than or equal to constraints with no lower bound.

```
public void getLBs(IloNumArray val, const IloIntVarArray vars) const
```

For each element of the array `vars`, this method puts the lower bound at the current node into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution values from `getValues` may violate these bounds at a node where a new incumbent has been found because the bounds are tightened when an incumbent is found.

Unbounded Variables

If a variable lacks a lower bound, then `getLBs` returns a value greater than or equal to `-IloInfinity` for greater than or equal to constraints with no lower bound.

```
public void getLBs(IloNumArray val, const IloNumVarArray vars) const
```

This method puts the lower bound at the current node of each element of the array `vars` into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution values from `getValues` may violate these bounds at a node where a new incumbent has been found because the bounds are tightened when an incumbent is found.

Unbounded Variables

If a variable lacks a lower bound, then `getLBs` returns a value greater than or equal to `-IloInfinity` for greater than or equal to constraints with no lower bound.

```
public NodeData * getNodeData() const
```

This method retrieves the `NodeData` object that may have previously been assigned to the current node by the user with the method `IloCplex::BranchCallbackI::makeBranch`. If no data object has been assigned to the current node, 0 (zero) will be returned.

```
public IloNum getObjValue() const
```

This method returns the objective value of the solution of the relaxation at the current node.

If you need the object representing the objective itself, consider the method `IloCplex::getObjective` instead.

```
public IloNum getSlack(const IloRange rng) const
```

This method returns the slack value for the constraint specified by `rng` in the solution of the relaxation at the current node.

```
public void getSlacks(IloNumArray val, const IloRangeArray con) const
```

For each of the constraints in the array of ranges `rngs`, this method puts the slack value in the solution of the relaxation at the current node into the corresponding element of the array `vals`.

```
public IloNum getUB(const IloIntVar var) const
```

This method returns the upper bound of the variable `var` at the current node. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution value from `getValue` may violate this bound at a node where a new incumbent has been found because the bound is tightened when an incumbent is found.

Unbounded Variables

If a variable lacks an upper bound, then `getUB` returns a value less than or equal to `IloInfinity` for less than or equal to constraints with no lower bound.

```
public IloNum getUB(const IloNumVar var) const
```

This method returns the upper bound of the variable `var` at the current node. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution value from `getValue` may violate this bound at a node where a new incumbent has been found because the bound is tightened when an incumbent is found.

Unbounded Variables

If a variable lacks an upper bound, then `getUB` returns a value less than or equal to `IloInfinity` for less than or equal to constraints with no lower bound.

```
public void getUBs(IloNumArray val, const IloIntVarArray vars) const
```

For each element in the array `vars`, this method puts the upper bound at the current node into the corresponding element of the array `vals`. The bounds are those in the relaxation at the current node. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution values from `getValues` may violate these bounds at a node where a new incumbent has been found because the bounds are tightened when an incumbent is found.

Unbounded Variables

If a variable lacks an upper bound, then `getUBs` returns a value less than or equal to `IloInfinity` for less than or equal to constraints with no lower bound.

```
public void getUBs(IloNumArray val, const IloNumVarArray vars) const
```

For each element in the array `vars`, this method puts the upper bound at the current node into the corresponding element of the array `vals`. The bounds are those in the relaxation at the current node. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes. The corresponding solution values from `getValues` may violate these bounds at a node where a new incumbent has been found because the bounds are tightened when an incumbent is found.

Unbounded Variables

If a variable lacks an upper bound, then `getUBs` returns a value less than or equal to `IloInfinity` for less than or equal to constraints with no lower bound.

```
public IloNum getUpPseudoCost(const IloIntVar var) const
```

This method returns the current pseudo cost for branching upward on the variable `var`.

```
public IloNum getUpPseudoCost(const IloNumVar var) const
```

This method returns the current pseudo cost for branching upward on the variable `var`.

```
public IloNum getValue(const IloIntVar var) const
```

This method returns the value of the variable `var` in the solution of the relaxation at the current node.

```
public IloNum getValue(const IloNumVar var) const
```

This method returns the value of the variable `var` in the solution of the relaxation at the current node.

```
public IloNum getValue(const IloExprArg expr) const
```

This method returns the value of the expression `expr` in the solution of the relaxation at the current node.

```
public void getValues(IloNumArray val, const IloIntVarArray vars) const
```

For each variable in the array `vars`, this method puts the value in the solution of the relaxation at the current node into the corresponding element of the array `vals`.

```
public void getValues(IloNumArray val, const IloNumVarArray vars) const
```

For each variable in the array `vars`, this method puts the value in the solution of the relaxation at the current node into the corresponding element of the array `vals`.

```
public IloBool isSOSFeasible(const IloSOS2 sos2) const
```

This method returns `IloTrue` if the solution of the LP at the current node is SOS feasible for the special ordered set specified in its argument. The SOS set passed as a parameter to this method may be of type 2. See the *CPLEX User's Manual* for more explanation of types of special ordered sets.

```
public IloBool isSOSFeasible(const IloSOS1 sos1) const
```

This method returns `IloTrue` if the solution of the LP at the current node is SOS feasible for the special ordered set specified in its argument. The SOS set passed as a parameter to this method may be of type 1. See the *CPLEX User's Manual* for more explanation about these types of special ordered sets.

Inner Enumerations

Enumeration IntegerFeasibility

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::ControlCallbackI::IntegerFeasibility` is an enumeration limited in scope to the class `IloCplex::ControlCallbackI`. This enumeration is used by `IloCplex::ControlCallbackI::getFeasibility` to represent the integer feasibility of a variable or SOS in the current node solution:

- `Feasible` specifies the variable or SOS is integer feasible.
- `ImpliedFeasible` specifies the variable or SOS has been presolved out. It will be feasible when all other integer variables or SOS are integer feasible.
- `Infeasible` specifies the variable or SOS is integer infeasible.

See Also: IloCplex, ControlCallbackI::IntegerFeasibilityArray

Fields:

`ImpliedInfeasible = -1`	
`Feasible = CPX_INTEGER_FEASIBLE`	Not applicable = CPX_INTEGER_FEASIBLE
`Infeasible = CPX_INTEGER_INFEASIBLE`	= CPX_INTEGER_INFEASIBLE
`ImpliedFeasible = CPX_IMPLIED_INTEGER_FEASIBLE`	= CPX_IMPLIED_INTEGER_FEASIBLE

Inner Typdefs

Typedef IntegerFeasibilityArray

Definition file: ilcplex/ilocplexi.h

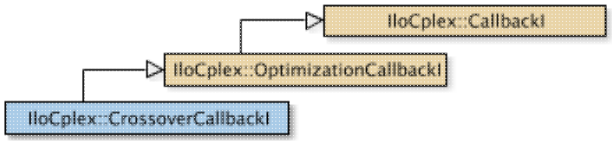
```
IloArray< IntegerFeasibility > IntegerFeasibilityArray
```

This type defines an array-type for `IloCplex::ControlCallbackI::IntegerFeasibility`. The fully qualified name of an integer feasibility array is `IloCplex::ControlCallbackI::IntegerFeasibility::Array`.

See Also: IloCplex, IloCplex::ControlCallbackI, ControlCallbackI::IntegerFeasibility

Class IloCplex::CrossoverCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::CrossoverCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a problem by means of the barrier optimizer with the crossover option. An instance of `IloCplex` calls this callback regularly during crossover. For details about the crossover option, see the *CPLEX User's Manual*.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::OptimizationCallbackI, ILOCROSSOVERCALLBACK0

Constructor Summary	
`protected`	`CrossoverCallbackI(IloEnv env)`

Method Summary	
`public IloInt`	`getNdualExchanges() const`
`public IloInt`	`getNdualPushes() const`
`public IloInt`	`getNprimalExchanges() const`
`public IloInt`	`getNprimalPushes() const`
`public IloInt`	`getNsuperbasics() const`

Inherited Methods from `OptimizationCallbackI`
`getModel, getNcols, getNQCs, getNrows`

Inherited Methods from `CallbackI`
`abort, duplicateCallback, getEndTime, getEnv, main`

Constructors

`protected` **`CrossoverCallbackI`**`(IloEnv env)`

This constructor creates a callback for use in an application with the crossover option of the barrier optimizer.

Methods

`public IloInt` **`getNdualExchanges`**`() const`

This method returns the number of dual exchange operations executed so far during crossover by the instance of `IloCplex` that executes the invoking callback.

```
public IloInt getNdualPushes() const
```

This method returns the number of dual push operations executed so far during crossover by the instance of `IloCplex` that executes the invoking callback.

```
public IloInt getNprimalExchanges() const
```

This method returns the number of primal exchange operations executed so far during crossover by the instance of `IloCplex` that executes the invoking callback.

```
public IloInt getNprimalPushes() const
```

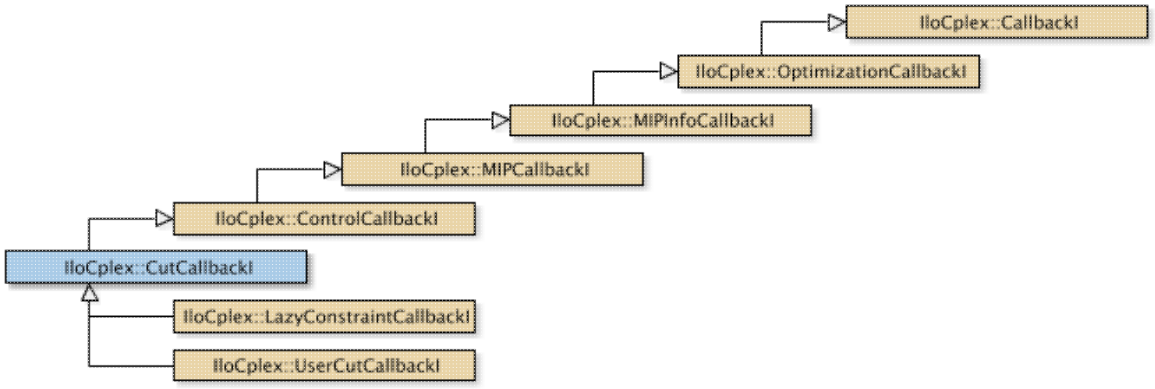
This method returns the number of primal push operations executed so far during crossover by the instance of `IloCplex` that executes the invoking callback.

```
public IloInt getNsuperbasics() const
```

This method returns the number of super basics currently present in the basis being generated with crossover by the instance of `IloCplex` that executes the invoking callback.

Class IloCplex::CutCallbackI

Definition file: ilcplex/ilocplexi.h



Note

This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::CutCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). This class offers a method to add a local or global cut to the current node LP subproblem from a user-written callback. CPLEX also calls the callback when comparing an integer feasible solution, including one provided by a MIP start before any nodes exist, against lazy constraints. More than one cut can be added in this callback by calling the method `add` or `addLocal` multiple times. All added cuts must be linear.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI, IloCplex::ControlCallbackI, IloCplex::OptimizationCallbackI, ILOCUTCALLBACK0

Constructor Summary	
`protected`	`CutCallbackI(IloEnv env)`

Method Summary	
`public IloConstraint`	`add(IloConstraint con)`
`public IloConstraint`	`addLocal(IloConstraint con)`

Inherited Methods from `ControlCallbackI`
`getDownPseudoCost, getDownPseudoCost, getFeasibilities, getFeasibilities, getFeasibility, getFeasibility, getFeasibility, getFeasibility, getLB, getLB, getLBs, getLBs, getNodeData, getObjValue, getSlack, getSlacks, getUB, getUB, getUBs, getUBs, getUpPseudoCost, getUpPseudoCost, getValue, getValue, getValue, getValues, getValues, isSOSFeasible, isSOSFeasible`

Inherited Methods from `MIPCallbackI`

getNcliques, getNcovers, getNcuts, getNdisjunctiveCuts, getNflowCovers, getNflowPaths, getNfractionalCuts, getNGUBcovers, getNimpliedBounds, getNMIRs, getNzeroHalfCuts, getObjCoef, getObjCoef, getObjCoefs, getObjCoefs, getUserThreads

Inherited Methods from `MIPInfoCallbackI`
getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent

Inherited Methods from `OptimizationCallbackI`
getModel, getNcols, getNQCs, getNrows

Inherited Methods from `CallbackI`
abort, duplicateCallback, getEndTime, getEnv, main

Constructors

```
protected CutCallbackI(IloEnv env)
```

This constructor creates a callback for use in an application with a user-defined cut to solve a MIP.

Methods

```
public IloConstraint add(IloConstraint con)
```

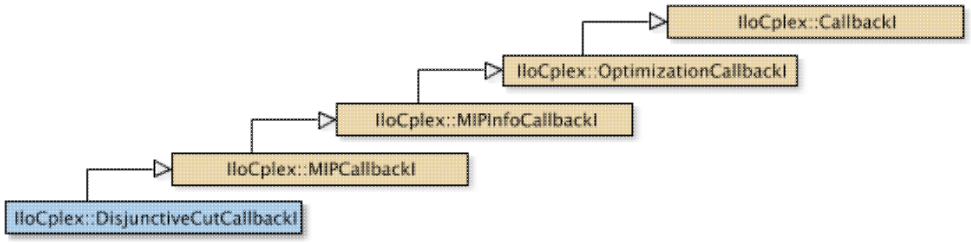
This method adds a cut to the current node LP subproblem for the constraint specified by `con`. This cut must be globally valid; it will not be removed by backtracking or any other means during the search. The added cut must be linear.

```
public IloConstraint addLocal(IloConstraint con)
```

This method adds a local cut to the current node LP subproblem for the constraint specified by `con`. `IloCplex` will manage the local cut in such a way that it will be active only when processing nodes of this subtree. The added cut must be linear.

Class IloCplex::DisjunctiveCutCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::DisjunctiveCutCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). This class offers a method to check on the progress of the generation of disjunctive cuts.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI, IloCplex::OptimizationCallbackI, ILODISJUNCTIVECUTCALLBACK0

Constructor Summary	
`protected`	`DisjunctiveCutCallbackI(IloEnv env)`

Method Summary	
`public IloNum`	`getProgress() const`

Inherited Methods from `MIPCallbackI`
`getNcliques, getNcovers, getNcuts, getNdisjunctiveCuts, getNflowCovers, getNflowPaths, getNfractionalCuts, getNGUBcovers, getNimpliedBounds, getNMIRs, getNzeroHalfCuts, getObjCoef, getObjCoef, getObjCoefs, getObjCoefs, getUserThreads`

Inherited Methods from `MIPInfoCallbackI`
`getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent`

Inherited Methods from `OptimizationCallbackI`
`getModel, getNcols, getNQCs, getNrows`

Inherited Methods from `CallbackI`
`abort, duplicateCallback, getEndTime, getEnv, main`

Constructors

`protected` **`DisjunctiveCutCallbackI`**`(IloEnv env)`

This constructor creates a callback for use in an application where disjunctive cuts are generated.

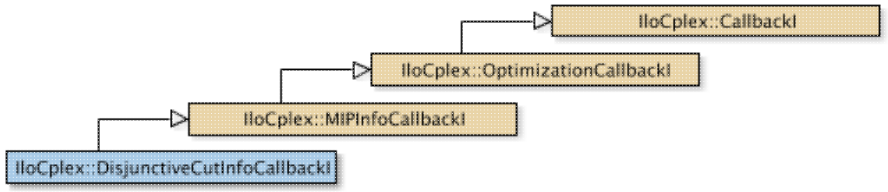
Methods

`public IloNum` **`getProgress`**`() const`

This method returns the fraction of completion of the disjunctive cut generation pass.

Class IloCplex::DisjunctiveCutInfoCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::DisjunctiveCutInfoCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). This class offers a method to check on the progress of the generation of disjunctive cuts.

User-written callbacks of this class are compatible with MIP dynamic search.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPInfoCallbackI, IloCplex::OptimizationCallbackI, ILODISJUNCTIVECUTINFOCALLBACK0

Constructor Summary	
protected	DisjunctiveCutInfoCallbackI(IloEnv env)

Method Summary	
public IloNum	getProgress() const

Inherited Methods from **MIPInfoCallbackI**
getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent

Inherited Methods from **OptimizationCallbackI**
getModel, getNcols, getNQCs, getNrows

Inherited Methods from **CallbackI**
abort, duplicateCallback, getEndTime, getEnv, main

Constructors

protected **DisjunctiveCutInfoCallbackI**(IloEnv env)

This constructor creates a callback for use in an application where disjunctive cuts are generated.

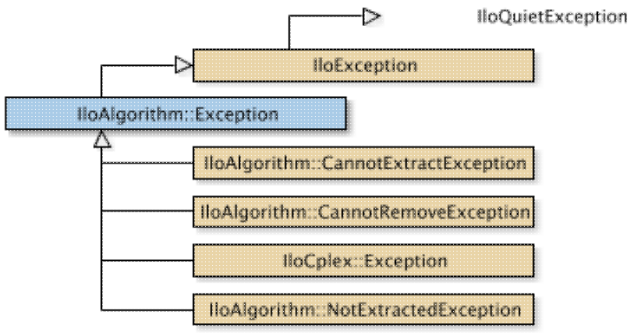
Methods

```
public IloNum getProgress() const
```

This method returns the fraction of completion of the disjunctive cut generation pass.

Class IloAlgorithm::Exception

Definition file: ilconcert/iloalg.h



The base class of exceptions thrown by classes derived from IloAlgorithm.
IloAlgorithm is the base class of algorithms in Concert Technology.

The class `IloAlgorithm::Exception`, derived from the class IloException, is the base class of exceptions thrown by classes derived from IloAlgorithm.

On platforms that support C++ exceptions, when exceptions are enabled, the member function IloAlgorithm::extract will throw an exception if you attempt to extract an unsuitable object from your model for an algorithm. An extractable object is unsuitable for an algorithm if there is no member function to extract the object from your model to that algorithm.

For example, an attempt to extract more than one objective into an algorithm that accepts only one objective will throw an exception.

Similarly, the member function IloAlgorithm::getValue will throw an exception if you attempt to access the value of a variable that has not yet been bound to a value.

See Also: IloAlgorithm, IloException

Constructor Summary	
public	Exception(const char * str)

Inherited Methods from `IloException`
end, getMessage

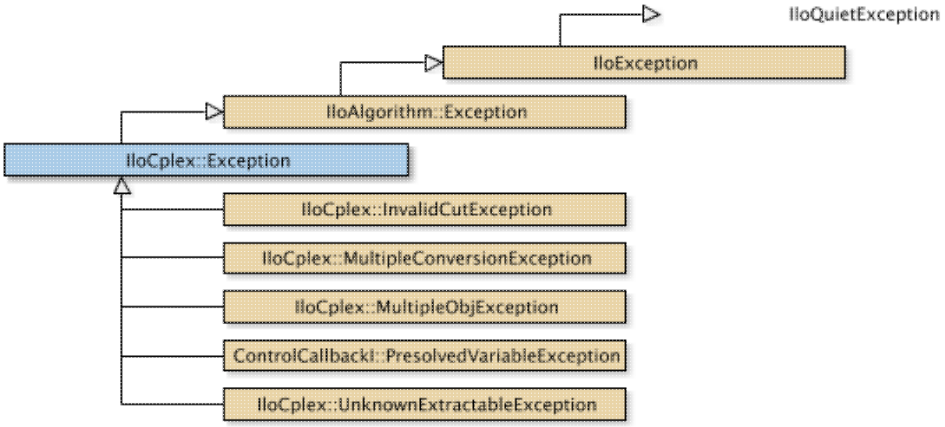
Constructors

public **Exception**(const char * str)

This constructor creates an exception thrown from a member of IloAlgorithm. The exception contains the message string str, which can be queried with the member function IloException::getMessage.

Class IloCplex::Exception

Definition file: ilcplex/ilocplexi.h



The class `IloCplex::Exception`, derived from the nested class `IloAlgorithm::Exception`, is the base class of exceptions thrown by classes derived from `IloCplex`.

Method Summary	
`public IloInt`	`getStatus() const`

Inherited Methods from `IloException`
`end, getMessage`

Methods

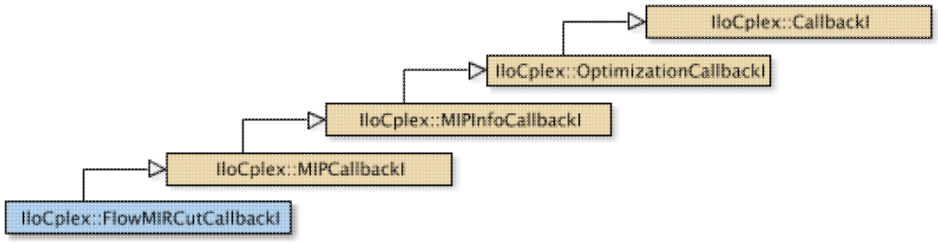
`public IloInt` **`getStatus`**`() const`

This method returns the CPLEX error code of an exception thrown by a member of `IloCplex`. These error codes are detailed in the reference manual as the group optim.cplex.errorcodes.

This method may also return negative values for subclasses of `IloCplex::Exception`, which are not listed in the reference manual. The exceptions listed in the reference manual are always thrown as instances of `IloCplex::Exception` and not as an instance of one of its derived classes.

Class IloCplex::FlowMIRCutCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::FlowMIRCutCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). This class offers a member function to check on the progress of the generation of Flow MIR cuts.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI, IloCplex::OptimizationCallbackI, ILOFLOWMIRCUTCALLBACK0

Constructor Summary	
`protected`	`FlowMIRCutCallbackI(IloEnv env)`

Method Summary	
`public IloNum`	`getProgress() const`

Inherited Methods from `MIPCallbackI`
`getNcliques, getNcovers, getNcuts, getNdisjunctiveCuts, getNflowCovers, getNflowPaths, getNfractionalCuts, getNGUBcovers, getNimpliedBounds, getNMIRs, getNzeroHalfCuts, getObjCoef, getObjCoef, getObjCoefs, getObjCoefs, getUserThreads`

Inherited Methods from `MIPInfoCallbackI`
`getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent`

Inherited Methods from `OptimizationCallbackI`
`getModel, getNcols, getNQCs, getNrows`

Inherited Methods from `CallbackI`
`abort, duplicateCallback, getEndTime, getEnv, main`

Constructors

```
protected FlowMIRCutCallbackI(IloEnv env)
```

This constructor creates a callback for use in an application where flow MIR cuts are generated.

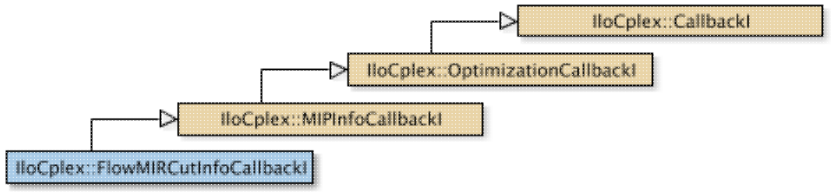
Methods

```
public IloNum getProgress() const
```

This method returns the fraction of completion of the cut generation pass for FlowMIR cuts.

Class IloCplex::FlowMIRCutInfoCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::FlowMIRCutInfoCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). This class offers a member function to check on the progress of the generation of Flow MIR cuts.

User-written callbacks of this class are compatible with MIP dynamic search.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPInfoCallbackI, IloCplex::OptimizationCallbackI, ILOFLOWMIRCUTINFOCALLBACK0

Constructor Summary	
protected	FlowMIRCutInfoCallbackI(IloEnv env)

Method Summary	
public IloNum	getProgress() const

Inherited Methods from **MIPInfoCallbackI**
getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent

Inherited Methods from **OptimizationCallbackI**
getModel, getNcols, getNQCs, getNrows

Inherited Methods from **CallbackI**
abort, duplicateCallback, getEndTime, getEnv, main

Constructors

protected **FlowMIRCutInfoCallbackI**(IloEnv env)

This constructor creates a callback for use in an application where flow MIR cuts are generated.

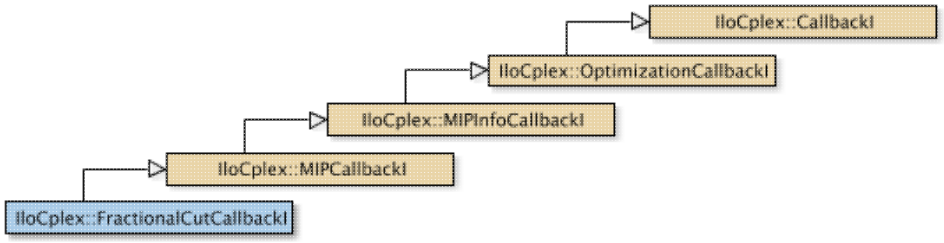
Methods

```
public IloNum getProgress() const
```

This method returns the fraction of completion of the cut generation pass for FlowMIR cuts.

Class IloCplex::FractionalCutCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::FractionalCutCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). This class offers a method to check on the progress of the generation of fractional cuts.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI, IloCplex::OptimizationCallbackI, ILOFRACTIONALCUTCALLBACK0

Constructor Summary	
protected	FractionalCutCallbackI(IloEnv env)

Method Summary	
public IloNum	getProgress() const

Inherited Methods from MIPCallbackI
getNcliques, getNcovers, getNcuts, getNdisjunctiveCuts, getNflowCovers, getNflowPaths, getNfractionalCuts, getNGUBcovers, getNimpliedBounds, getNMIRs, getNzeroHalfCuts, getObjCoef, getObjCoef, getObjCoefs, getObjCoefs, getUserThreads

Inherited Methods from MIPInfoCallbackI
getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent

Inherited Methods from OptimizationCallbackI
getModel, getNcols, getNQCs, getNrows

Inherited Methods from CallbackI
abort, duplicateCallback, getEndTime, getEnv, main

Constructors

`protected` **`FractionalCutCallbackI`**`(IloEnv env)`

This constructor creates a callback for use in an application where fractional cuts are generated.

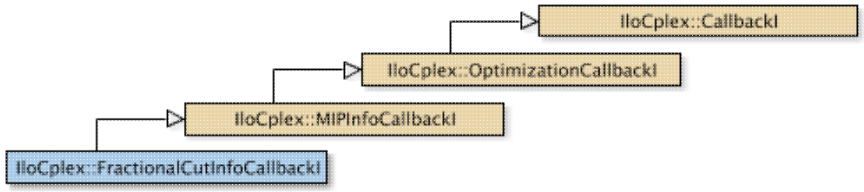
Methods

`public IloNum` **`getProgress`**`() const`

This method returns the fraction of completion of the fractional cut generation pass.

Class IloCplex::FractionalCutInfoCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::FractionalCutInfoCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). This class offers a method to check on the progress of the generation of fractional cuts.

User-written callbacks of this class are compatible with MIP dynamic search.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPInfoCallbackI, IloCplex::OptimizationCallbackI, ILOFRACTIONALCUTINFOCALLBACK0

Constructor Summary	
`protected`	`FractionalCutInfoCallbackI(IloEnv env)`

Method Summary	
`public IloNum`	`getProgress() const`

Inherited Methods from `MIPInfoCallbackI`
`getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent`

Inherited Methods from `OptimizationCallbackI`
`getModel, getNcols, getNQCs, getNrows`

Inherited Methods from `CallbackI`
`abort, duplicateCallback, getEndTime, getEnv, main`

Constructors

`protected` **`FractionalCutInfoCallbackI`**`(IloEnv env)`

This constructor creates a callback for use in an application where fractional cuts are generated.

Methods

```
public IloNum getProgress() const
```

This method returns the fraction of completion of the fractional cut generation pass.

Class IloCplex::Goal

Definition file: ilcplex/ilocplexi.h

Note

This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

Goals can be used to control the branch-and-cut search in `IloCplex`. Goals are implemented in the class `IloCplex::GoalI`. This is the handle class for CPLEX goals.

Goal objects are reference-counted. This means every instance of `IloCplex::GoalI` keeps track about how many handle objects refer to it. If this number drops to 0 (zero) the `IloCplex::GoalI` object is automatically deleted. As a consequence, whenever you deal with a goal, you must keep a handle object around, rather than only a pointer to the implementation object. Otherwise, you risk ending up with a pointer to an implementation object that has already been deleted.

See *Goals* among the Concepts in this manual. See also goals in the *CPLEX User's Manual*.

Constructor and Destructor Summary	
public	`Goal(IloCplex::GoalBaseI * goalI)`
public	`Goal(const Goal & goal)`
public	`Goal()`
public	`Goal(IloConstraint cut)`
public	`Goal(IloConstraintArray cut)`

Method Summary	
public Goal	`operator=(const Goal & goal)`

Constructors and Destructors

`public` **`Goal`**`(IloCplex::GoalBaseI * goalI)`

Creates a new goal from a pointer to the implementation object.

`public` **`Goal`**`(const Goal & goal)`

This is the copy constructor of the goal.

`public` **`Goal`**`()`

Creates a 0 goal handle, that is, a goal with a 0 implementation object pointer. This is also referred to as an empty goal.

```
public Goal(IloConstraint cut)
```

Creates a new goal that will add the constraint `cut` as a local cut to the node where the goal is executed. As a local cut, the constraint will be active only in the subtree rooted at the node where the goal was executed. The lifetime of the constraint passed to a goal is tied to the lifetime of the Goal. That is, the constraint's method `end` is called when the goal's implementation object is deleted. As a consequence, the method `end` must not be called for constraints passed to this constructor explicitly.

```
public Goal(IloConstraintArray cut)
```

Creates a new goal that adds the constraints given in the array `cut` as local cuts to the node where the goal is executed. As local cuts, the constraints will be active only in the subtree rooted at the node where the goal was executed. The lifetime of the constraints and the array passed to a goal is tied to the lifetime of the Goal. That is, the constraint's method `end` is called when the goal's implementation object is deleted. As a consequence, method `end` must not be called for the constraints and the array passed to this constructor explicitly.

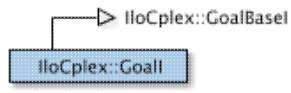
Methods

```
public Goal operator=(const Goal & goal)
```

This is the assignment operator. It increases the reference count of the implementation object of `goal`. If the invoking handle referred to an implementation object before the assignment operation, its reference count is decreased. If thereby the reference count becomes 0, the implementation object is deleted.

Class IloCplex::GoalI

Definition file: ilcplex/ilocplexi.h



Note

This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

Goals can be used to control the branch-and-cut search in `IloCplex`. Goals are implemented in subclasses of the class `IloCplex::GoalI`. This is the base class for user-written implementation classes of CPLEX goals.

To implement your own goal you need to create a subclass of `IloCplex::GoalI` and implement its pure virtual methods `execute` and `duplicateGoal`. You may use one of the `ILOCPLEXGOAL0` macros to assist you in doing so. After implementing your goal class, you use an instance of the class by passing it to the `solve` method when solving the model.

The method `duplicateGoal` may be called by `IloCplex` to create copies of a goal when needed for parallel branch-and-cut search. Thus the implementation of this method must create and return an exact copy of the invoked object itself.

The method `execute` controls the branch-and-cut search of `IloCplex` by the goal it returns. When `IloCplex` processes a node, it pops the top goal from the node's goal stack and calls method `execute` of that goal. It continues executing the top goal from the stack until the node is deactivated or the goal stack is empty. If the goal stack is empty, `IloCplex` proceeds with the built-in search strategy for the subtree rooted at the current node.

The class `IloCplex::GoalI` provides several methods for querying information about the current node. The method `execute` controls how to proceed with the branch-and-cut search via the goal it returns. The returned goal, unless it is the 0 goal, is pushed on the goal stack and will thus be executed next.

See also the chapter about goals in the *CPLEX User's Manual*.

Constructor Summary	
`public`	`GoalI(IloEnv env)`

Method Summary	
`public void`	`abort()`
`public static IloCplex::Goal`	`AndGoal(IloCplex::Goal goal1, IloCplex::Goal goal2, IloCplex::Goal goal3, IloCplex::Goal goal4, IloCplex::Goal goal5, IloCplex::Goal goal6)`
`public static IloCplex::Goal`	`AndGoal(IloCplex::Goal goal1, IloCplex::Goal goal2, IloCplex::Goal goal3, IloCplex::Goal goal4, IloCplex::Goal goal5)`
`public static IloCplex::Goal`	`AndGoal(IloCplex::Goal goal1, IloCplex::Goal goal2, IloCplex::Goal goal3, IloCplex::Goal goal4)`
`public static IloCplex::Goal`	

	AndGoal(IloCplex::Goal goal1, IloCplex::Goal goal2, IloCplex::Goal goal3)
public static IloCplex::Goal	AndGoal(IloCplex::Goal goal1, IloCplex::Goal goal2)
public static IloCplex::Goal	BranchAsCplexGoal(IloEnv env)
public virtual IloCplex::Goal	duplicateGoal()
public virtual IloCplex::Goal	execute()
public static IloCplex::Goal	FailGoal(IloEnv env)
public IloNum	getBestObjValue() const
public IloNum	getBranch(IloNumVarArray vars, IloNumArray bounds, IloCplex::BranchDirectionArray dirs, IloInt i) const
public GoalI::BranchType	getBranchType() const
public IloNum	getCutoff() const
public IloCplex::BranchDirection	getDirection(const IloIntVar var)
public IloCplex::BranchDirection	getDirection(const IloNumVar var)
public IloNum	getDownPseudoCost(const IloIntVar var) const
public IloNum	getDownPseudoCost(const IloNumVar var) const
public IloEnv	getEnv() const
public void	getFeasibilities(GoalI::IntegerFeasibilityArray stats, const IloIntVarArray vars) const
public void	getFeasibilities(GoalI::IntegerFeasibilityArray stats, const IloNumVarArray vars) const
public GoalI::IntegerFeasibility	getFeasibility(const IloSOS2 sos) const
public GoalI::IntegerFeasibility	getFeasibility(const IloSOS1 sos) const
public GoalI::IntegerFeasibility	getFeasibility(const IloIntVar var) const
public GoalI::IntegerFeasibility	getFeasibility(const IloNumVar var) const
public IloNum	getIncumbentObjValue() const
public IloNum	getIncumbentValue(const IloIntVar var) const
public IloNum	getIncumbentValue(const IloNumVar var) const
public void	getIncumbentValues(IloNumArray val, const IloIntVarArray vars) const
public void	getIncumbentValues(IloNumArray val, const IloNumVarArray vars) const
public IloNum	getLB(const IloIntVar var) const
public IloNum	getLB(const IloNumVar var) const
public void	getLBs(IloNumArray vals, const IloIntVarArray vars) const
public void	getLBs(IloNumArray vals, const IloNumVarArray vars) const
public IloNum	getMIPRelativeGap() const
public IloModel	getModel() const
public IloInt	getMyThreadNum() const
public IloInt	getNbranches() const
public IloInt	getNcliques() const

public IloInt	getNcols() const
public IloInt	getNcovers() const
public IloInt	getNcuts(IloCplex::CutType which) const
public IloInt	getNdisjunctiveCuts() const
public IloInt	getNflowCovers() const
public IloInt	getNflowPaths() const
public IloInt	getNfractionalCuts() const
public IloInt	getNGUBcovers() const
public IloInt	getNimpliedBounds() const
public IloInt	getNiterations() const
public IloInt	getNMIRs() const
public IloInt	getNnodes() const
public IloInt	getNremainingNodes() const
public IloInt	getNrows() const
public IloInt	getNzeroHalfCuts() const
public IloNum	getObjCoef(const IloIntVar var) const
public IloNum	getObjCoef(const IloNumVar var) const
public void	getObjCoefs(IloNumArray vals, const IloIntVarArray vars) const
public void	getObjCoefs(IloNumArray vals, const IloNumVarArray vars) const
public IloNum	getObjValue() const
public IloNum	getPriority(const IloIntVar var) const
public IloNum	getPriority(const IloNumVar var) const
public IloNum	getSlack(const IloRange rng) const
public void	getSlacks(IloNumArray vals, const IloRangeArray rngs) const
public IloNum	getUB(const IloIntVar var) const
public IloNum	getUB(const IloNumVar var) const
public void	getUBs(IloNumArray vals, const IloIntVarArray vars) const
public void	getUBs(IloNumArray vals, const IloNumVarArray vars) const
public IloNum	getUpPseudoCost(const IloIntVar var) const
public IloNum	getUpPseudoCost(const IloNumVar var) const
public IloInt	getUserThreads() const
public IloNum	getValue(const IloIntVar var) const
public IloNum	getValue(const IloNumVar var) const
public IloNum	getValue(const IloExpr expr) const
public void	getValues(IloNumArray vals, const IloIntVarArray vars) const
public void	getValues(IloNumArray vals, const IloNumVarArray vars) const
public static IloCplex::Goal	GlobalCutGoal(IloConstraintArray con)

public static IloCplex::Goal	GlobalCutGoal(IloConstraint con)
public IloBool	hasIncumbent() const
public IloBool	isIntegerFeasible() const
public IloBool	isSOSFeasible(const IloSOS2 sos2) const
public IloBool	isSOSFeasible(const IloSOS1 sos1) const
public static IloCplex::Goal	OrGoal(IloCplex::Goal goal1, IloCplex::Goal goal2, IloCplex::Goal goal3, IloCplex::Goal goal4, IloCplex::Goal goal5, IloCplex::Goal goal6)
public static IloCplex::Goal	OrGoal(IloCplex::Goal goal1, IloCplex::Goal goal2, IloCplex::Goal goal3, IloCplex::Goal goal4, IloCplex::Goal goal5)
public static IloCplex::Goal	OrGoal(IloCplex::Goal goal1, IloCplex::Goal goal2, IloCplex::Goal goal3, IloCplex::Goal goal4)
public static IloCplex::Goal	OrGoal(IloCplex::Goal goal1, IloCplex::Goal goal2, IloCplex::Goal goal3)
public static IloCplex::Goal	OrGoal(IloCplex::Goal goal1, IloCplex::Goal goal2)
public static IloCplex::Goal	SolutionGoal(const IloIntVarArray vars, const IloNumArray vals)
public static IloCplex::Goal	SolutionGoal(const IloNumVarArray vars, const IloNumArray vals)

Inner Enumeration
GoalI::BranchType
GoalI::IntegerFeasibility

Inner Typedef
GoalI::IntegerFeasibilityArray

Constructors

`public` **`GoalI`**`(IloEnv env)`

The goal constructor. It requires an instance of the same `IloEnv` as the `IloCplex` object with which to use the goal. The environment can later be queried by calling method `getEnv`.

Methods

`public void` **`abort`**`()`

Abort the optimization, that is, the execution of method `IloCplex::solve` currently in process.

```
public static IloCplex::Goal AndGoal(IloCplex::Goal goal1, IloCplex::Goal goal2)
public static IloCplex::Goal AndGoal(IloCplex::Goal goal1, IloCplex::Goal goal2,
IloCplex::Goal goal3, IloCplex::Goal goal4, IloCplex::Goal goal5, IloCplex::Goal
goal6)
public static IloCplex::Goal AndGoal(IloCplex::Goal goal1, IloCplex::Goal goal2,
```

```
IloCplex::Goal goal3, IloCplex::Goal goal4, IloCplex::Goal goal5)
public static IloCplex::Goal AndGoal(IloCplex::Goal goal1, IloCplex::Goal goal2,
IloCplex::Goal goal3, IloCplex::Goal goal4)
public static IloCplex::Goal AndGoal(IloCplex::Goal goal1, IloCplex::Goal goal2,
IloCplex::Goal goal3)
```

The static methods `AndGoal` all return a goal that pushes the goals passed as parameters onto the goal stack in reverse order. As a consequence, the goals will be executed in the order they are passed as parameters to the `AndGoal` function.

```
public static IloCplex::Goal BranchAsCplexGoal(IloEnv env)
```

This static function returns a goal that creates the same branches as the currently selected built-in branch strategy of `IloCplex` would choose at the current node. This goal allows you to proceed with the `IloCplex` search strategy, but keeps the search under goal control, thereby giving you the option to intervene at any point.

This goal is also important when you use node evaluators while you use a built-in branching strategy.

For example, consider the `execute` method of a goal starting like this:

```
if (!isIntegerFeasible())
    return AndGoal(BranchAsCplexGoal(getEnv()), this);

// do something
```

It would `do something` only when `IloCplex` found a solution it considers to be a candidate for a new incumbent. Note there is a test of integer feasibility before returning `BranchAsCplexGoal`. Without the test, `BranchAsCplex` would be executed for a solution `IloCplex` considers to be feasible, but `IloCplex` would not know how to branch on it. An endless loop would result.

```
public virtual IloCplex::Goal duplicateGoal()
```

This virtual method must be implemented by the user. It must return a copy of the invoking goal object. This method may be called by `IloCplex` when doing parallel branch-and-cut search.

```
public virtual IloCplex::Goal execute()
```

This virtual method must be implemented by the user to specify the logic of the goal. The instance of `IloCplex::Goal` returned by this method will be added to the goal stack of the node where the invoking goal is being executed for further execution.

```
public static IloCplex::Goal FailGoal(IloEnv env)
```

This static method creates a goal that fails. That means that the branch where the goal is executed will be pruned or, equivalently, the search is discontinued at that node and the node is discarded.

```
public IloNum getBestObjValue() const
```

This method accesses the currently best known bound of all the remaining open nodes in a branch-and-cut tree.

It is computed for a minimization problem as the minimum objective function value of all remaining unexplored nodes. Similarly, it is computed for a maximization problem as the maximum objective function value of all remaining unexplored nodes.

For a regular MIP optimization, this value is also the best known bound on the optimal solution value of the MIP problem. In fact, when a problem has been solved to optimality, this value matches the optimal solution value.

However, for the method `populate`, the value can also exceed the optimal solution value if CPLEX has already solved the model to optimality but continues to search for additional solutions.

```
public IloNum getBranch(IloNumVarArray vars, IloNumArray bounds,
IloCplex::BranchDirectionArray dirs, IloInt i) const
```

This method accesses branching information for the `i`-th branch that the invoking instance of `IloCplex` is about to create. The parameter `i` must be between 0 (zero) and `getNbranches - 1`; that is, it must be a valid index of a branch; normally, it will be zero or one.

A branch is normally defined by a set of variables and the bounds for these variables. Branches that are more complex cannot be queried. The return value is the node estimate for that branch.

- The parameter `vars` contains the variables for which new bounds will be set in the `i`-th branch.
- The parameter `bounds` contains the new bounds for the variables listed in `vars`; that is, `bounds[j]` is the new bound for `vars[j]`.
- The parameter `dirs` specifies the branching direction for the variables in `vars`.

```
dir[j] == IloCplex::BranchUp
```

means that `bounds[j]` specifies a lower bound for `vars[j]`.

```
dirs[j] == IloCplex::BranchDown
```

means that `bounds[j]` specifies an upper bound for `vars[j]`.

```
public GoalI::BranchType getBranchType() const
```

This method returns the type of branching `IloCplex` is going to do for the current node.

```
public IloNum getCutoff() const
```

The method returns the current cutoff value. An instance of `IloCplex` uses the cutoff value (the value of the objective function of the subproblem at a node in the search tree) to decide when to prune nodes from the search tree (that is, when to cut off that node and discard the nodes beyond it). The cutoff value is updated whenever a new incumbent is found.

```
public IloCplex::BranchDirection getDirection(const IloIntVar var)
```

This method returns the branch direction previously assigned to variable `var` with the method `IloCplex::setDirection` or `IloCplex::setDirections`. If no direction has been assigned, `IloCplex::BranchGlobal` will be returned.

```
public IloCplex::BranchDirection getDirection(const IloNumVar var)
```

This method returns the branch direction previously assigned to variable `var` with the method `IloCplex::setDirection` or `IloCplex::setDirections`. If no direction has been assigned, `IloCplex::BranchGlobal` will be returned.

```
public IloNum getDownPseudoCost(const IloIntVar var) const
```

This method returns the current pseudo cost for branching downward on the variable `var`.

```
public IloNum getDownPseudoCost(const IloNumVar var) const
```

This method returns the current pseudo cost for branching downward on the variable `var`.

```
public IloEnv getEnv() const
```

Returns the instance of `IloEnv` passed to the constructor of the goal.

```
public void getFeasibilities(GoalI::IntegerFeasibilityArray stats, const
IloIntVarArray vars) const
```

This method considers whether each of the variables in the array `vars` is integer feasible, integer infeasible, or implied integer feasible and puts the status in the corresponding element of the array `stats`.

```
public void getFeasibilities(GoalI::IntegerFeasibilityArray stats, const
IloNumVarArray vars) const
```

This method considers whether each of the variables in the array `vars` is integer feasible, integer infeasible, or implied integer feasible and puts the status in the corresponding element of the array `stats`.

```
public GoalI::IntegerFeasibility getFeasibility(const IloSOS2 sos) const
```

This method specifies whether the SOS `sos` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public GoalI::IntegerFeasibility getFeasibility(const IloSOS1 sos) const
```

This method specifies whether the SOS `sos` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public GoalI::IntegerFeasibility getFeasibility(const IloIntVar var) const
```

This method specifies whether the variable `var` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public GoalI::IntegerFeasibility getFeasibility(const IloNumVar var) const
```

This method specifies whether the variable `var` is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```
public IloNum getIncumbentObjValue() const
```

This method returns the value of the objective function of the incumbent solution (that is, the best integer solution found so far). If there is no incumbent, this method throws an exception.

```
public IloNum getIncumbentValue(const IloIntVar var) const
```

This method returns the value of `var` in the incumbent solution. If there is no incumbent, this method throws an exception.

```
public IloNum getIncumbentValue(const IloNumVar var) const
```

This method returns the value of `var` in the incumbent solution. If there is no incumbent, this method throws an exception.

```
public void getIncumbentValues(IloNumArray val, const IloIntVarArray vars) const
```

Returns the value of each variable in the array `vars` with respect to the current incumbent solution, and it puts those values into the corresponding array `vals`. If there is no incumbent, this method throws an exception.

```
public void getIncumbentValues(IloNumArray val, const IloNumVarArray vars) const
```

Returns the value of each variable in the array `vars` with respect to the current incumbent solution, and it puts those values into the corresponding array `vals`. If there is no incumbent, this method throws an exception.

```
public IloNum getLB(const IloIntVar var) const
```

This method returns the lower bound of `var` in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

Unbounded Variables

If a variable lacks a lower bound, then `getLB` returns a value greater than or equal to `-IloInfinity` for greater than or equal to constraints with no lower bound.

```
public IloNum getLB(const IloNumVar var) const
```

This method returns the lower bound of `var` in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

Unbounded Variables

If a variable lacks a lower bound, then `getLB` returns a value greater than or equal to `-IloInfinity` for greater than or equal to constraints with no lower bound.

```
public void getLBs(IloNumArray vals, const IloIntVarArray vars) const
```

This method puts the lower bound in the current node relaxation of each element of the array `vars` into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

Unbounded Variables

If a variable lacks a lower bound, then `getLBs` returns a value greater than or equal to `-IloInfinity` for greater than or equal to constraints with no lower bound.

```
public void getLBs(IloNumArray vals, const IloNumVarArray vars) const
```

This method puts the lower bound in the current node relaxation of each element of the array `vars` into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

Unbounded Variables

If a variable lacks a lower bound, then `getLBs` returns a value greater than or equal to `-IloInfinity` for greater than or equal to constraints with no lower bound.

```
public IloNum getMIPRelativeGap() const
```

This method accesses the current relative objective gap.

For a **minimization** problem, this value is computed by

```
(bestinteger - bestobjective) / (1e-10 + |bestobjective|)
```

where `bestinteger` is the value returned by `IloCplex::GoalI::getIncumbentValue` and `bestobjective` is the value returned by `IloCplex::GoalI::getBestObjValue`.

For a **maximization** problem, the value is computed by:

```
(bestobjective - bestinteger) / (1e-10 + |bestobjective|)
```

```
public IloModel getModel() const
```

This method returns the model currently extracted for the instance of `IloCplex` where the invoking goal applies.

```
public IloInt getMyThreadNum() const
```

Returns the identifier of the parallel thread being currently executed. This number is between 0 (zero) and the value returned by the method `getUserThreads()-1`.

```
public IloInt getNbranches() const
```

This method returns the number of branches `IloCplex` is going to create at the current node.

```
public IloInt getNcliques() const
```

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of clique cuts that have been added to the model so far during the current optimization.

```
public IloInt getNcols() const
```

This method returns the number of columns in the current optimization model.

```
public IloInt getNcovers() const
```

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of cover cuts that have been added to the model so far during the current optimization.

```
public IloInt getNcuts(IloCplex::CutType which) const
```

Returns the total number of cuts of the type `which` that CPLEX has added to the model so far during the current optimization.

```
public IloInt getNdisjunctiveCuts() const
```

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of disjunctive cuts that have been added to the model so far during the current optimization.

```
public IloInt getNflowCovers() const
```

Returns the total number of flow cover cuts that have been added to the model so far during the current optimization.

```
public IloInt getNflowPaths() const
```

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of flow path cuts that have been added to the model so far during the current optimization.

```
public IloInt getNfractionalCuts() const
```

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of fractional cuts that have been added to the model so far during the current optimization.

```
public IloInt getNGUBcovers() const
```

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of GUB cover cuts that have been added to the model so far during the current optimization.

```
public IloInt getNimpliedBounds() const
```

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of implied bound cuts that have been added to the model so far during the current optimization.

```
public IloInt getNiterations() const
```

Returns the total number of iterations executed so far during the current optimization to solve the node relaxations.

```
public IloInt getNMIRs() const
```

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of MIR cuts that have been added to the model so far during the current optimization.

```
public IloInt getNnodes() const
```

This method returns the number of nodes already processed in the current optimization.

```
public IloInt getNremainingNodes() const
```

This method returns the number of nodes left to explore in the current optimization.

```
public IloInt getNrows() const
```

This method returns the number of rows in the current optimization model.

```
public IloInt getNzeroHalfCuts() const
```

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of zero-half cuts that have been added to the model so far during the current optimization.

```
public IloNum getObjCoef(const IloIntVar var) const
```

Returns the linear objective coefficient for `var` in the model currently being solved.

```
public IloNum getObjCoef(const IloNumVar var) const
```

Returns the linear objective coefficient for `var` in the model currently being solved.

```
public void getObjCoefs(IloNumArray vals, const IloIntVarArray vars) const
```

This method puts the linear objective coefficient of each of the variables in the array `vars` into the corresponding element of the array `vals`.

```
public void getObjCoefs(IloNumArray vals, const IloNumVarArray vars) const
```

This method puts the linear objective coefficient of each of the variables in the array `vars` into the corresponding element of the array `vals`.

```
public IloNum getObjValue() const
```

This method returns the objective value of the solution of the current node.

If you need the object representing the objective itself, consider the method `IloCplex::getObjective` instead.

```
public IloNum getPriority(const IloIntVar var) const
```

Returns the branch priority used for variable `var` in the current optimization.

```
public IloNum getPriority(const IloNumVar var) const
```

Returns the branch priority used for variable `var` in the current optimization.

```
public IloNum getSlack(const IloRange rng) const
```

This method returns the slack value for the constraint specified by `rng` in the solution of the current node relaxation.

```
public void getSlacks(IloNumArray vals, const IloRangeArray rngs) const
```

This method puts the slack value in the solution of the current node relaxation of each of the constraints in the array of ranges `rngs` into the corresponding element of the array `vals`.

```
public IloNum getUB(const IloIntVar var) const
```

This method returns the upper bound of the variable `var` in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

Unbounded Variables

If a variable lacks an upper bound, then `getUB` returns a value less than or equal to `IloInfinity` for less than or equal to constraints with no lower bound.

```
public IloNum getUB(const IloNumVar var) const
```

This method returns the upper bound of the variable `var` in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

Unbounded Variables

If a variable lacks an upper bound, then `getUB` returns a value less than or equal to `IloInfinity` for less than or equal to constraints with no lower bound.

```
public void getUBs(IloNumArray vals, const IloIntVarArray vars) const
```

This method puts the upper bound in the current node relaxation of each element of the array `vars` into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

Unbounded Variables

If a variable lacks an upper bound, then `getUBs` returns a value less than or equal to `IloInfinity` for less than or equal to constraints with no lower bound.

```
public void getUBs(IloNumArray vals, const IloNumVarArray vars) const
```

This method puts the upper bound in the current node relaxation of each element of the array `vars` into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

Unbounded Variables

If a variable lacks an upper bound, then `getUBs` returns a value less than or equal to `IloInfinity` for less than or equal to constraints with no lower bound.

```
public IloNum getUpPseudoCost(const IloIntVar var) const
```

This method returns the current pseudo cost for branching upward on the variable `var`.

```
public IloNum getUpPseudoCost(const IloNumVar var) const
```

This method returns the current pseudo cost for branching upward on the variable `var`.

```
public IloInt getUserThreads() const
```

This method returns the total number of parallel threads currently running.

```
public IloNum getValue(const IloIntVar var) const
```

This method returns the value of the variable `var` in the solution of the current node relaxation.

```
public IloNum getValue(const IloNumVar var) const
```

This method returns the value of the variable `var` in the solution of the current node relaxation.

```
public IloNum getValue(const IloExpr expr) const
```

This method returns the value of the expression `expr` in the solution of the current node relaxation.

```
public void getValues(IloNumArray vals, const IloIntVarArray vars) const
```

This method puts the current node relaxation solution value of each variable in the array `vars` into the corresponding element of the array `vals`.

```
public void getValues(IloNumArray vals, const IloNumVarArray vars) const
```

This method puts the current node relaxation solution value of each variable in the array `vars` into the corresponding element of the array `vals`.

```
public static IloCplex::Goal GlobalCutGoal(IloConstraintArray con)
```

This method creates a goal that when executed adds the constraints (provided in the paramter array `con`) as global cuts to the model. These global cuts must be valid for the entire model, not only for the current subtree. In other words, these global cuts will be respected at every node.

`IloCplex` takes over memory managment for the cuts passed to the method `GlobalCutGoal`. Thus `IloCplex` will call the method `end` as soon as it can be discarded after the goal executes. Calling `end` yourself or the constraints in the array `con` passed to method `GlobalCutGoal` or the array itself is an error and must be avoided.

```
public static IloCplex::Goal GlobalCutGoal(IloConstraint con)
```

This method creates a goal that when executed adds the constraint `con` (provided as a parameter) as global cuts to the model. These global cuts must be valid for the entire model, not only for the current subtree. In other words, these global cuts will be respected at every node.

`IloCplex` takes over memory managment for the cut passed to the method `GlobalCutGoal`. Thus `IloCplex` will call the method `end` as soon as it can be discarded after the goal executes. Calling `end` yourself for the constraint passed to method `GlobalCutGoal` is an error and must be avoided.

```
public IloBool hasIncumbent() const
```

This method returns `IloTrue` if an integer feasible solution has been found.

```
public IloBool isIntegerFeasible() const
```

This method returns `IloTrue` if the solution of the current node is integer feasible.

```
public IloBool isSOSFeasible(const IloSOS2 sos2) const
```

This method returns `IloTrue` if the solution of the current node is SOS feasible for the special ordered set specified in its argument. The SOS passed as a parameter to this method must be of type 2; the equivalent method for an SOS of type 1 is also available. See the *User's Manual* for more about these types of special ordered sets.

```
public IloBool isSOSFeasible(const IloSOS1 sos1) const
```

This method returns `IloTrue` if the solution of the current node is SOS feasible for the special ordered set specified in its argument. The SOS passed as a parameter to this method must be of type 1; the equivalent method for an SOS of type 2 is also available. See the *User's Manual* for more about these types of special ordered sets.

```
public static IloCplex::Goal OrGoal(IloCplex::Goal goal1, IloCplex::Goal goal2)
public static IloCplex::Goal OrGoal(IloCplex::Goal goal1, IloCplex::Goal goal2,
IloCplex::Goal goal3, IloCplex::Goal goal4, IloCplex::Goal goal5, IloCplex::Goal
goal6)
public static IloCplex::Goal OrGoal(IloCplex::Goal goal1, IloCplex::Goal goal2,
IloCplex::Goal goal3, IloCplex::Goal goal4, IloCplex::Goal goal5)
public static IloCplex::Goal OrGoal(IloCplex::Goal goal1, IloCplex::Goal goal2,
IloCplex::Goal goal3, IloCplex::Goal goal4)
public static IloCplex::Goal OrGoal(IloCplex::Goal goal1, IloCplex::Goal goal2,
IloCplex::Goal goal3)
```

The static methods `OrGoal` all return a goal that creates as many branches (or, equivalently, subproblems) as there are parameters. Each of the subnodes will be initialized with the remaining goal stack of the current node. In addition, the goal parameter will be pushed on the goal stack of the corresponding subgoal. If more than six branches need to be created, instances of `OrGoal` can be combined.

```
public static IloCplex::Goal SolutionGoal(const IloIntVarArray vars, const
```

```
IloNumArray vals)
```

This static method creates and returns a goal that attempts to inject a solution specified by setting the variables listed in array `vars` to the corresponding values listed in the array `vals`.

`IloCplex` will not blindly accept such a solution as a new incumbent. Instead, it will make sure that this solution is compatible with both the model and the goals. When checking feasibility with goals, it checks feasibility with both the goals that have already been executed and the goals that are still on the goal stack. Thus, in particular, `IloCplex` will reject any solution that is not compatible with the branching that has been done so far.

`IloCplex` takes over memory managment for arrays `vars` and `vals` passed to `SolutionGoal`. Thus `IloCplex` will call method `end` for these arrays as soon as they can be discarded. Calling `end` for the arrays passed to `SolutionGoal` is an error and must be avoided.

```
public static IloCplex::Goal SolutionGoal(const IloNumVarArray vars, const
IloNumArray vals)
```

This static method creates and returns a goal that attempts to inject a solution specified by setting the variables listed in array `vars` to the corresponding values listed in the array `vals`.

`IloCplex` will not blindly accept such a solution as a new incumbent. Instead, it will make sure that this solution is compatible with both the model and the goals. When checking feasibility with goals, it checks feasibility with both the goals that have already been executed and the goals that are still on the goal stack. Thus, in particular, `IloCplex` will reject any solution that is not compatible with the branching that has been done so far.

`IloCplex` takes over memory managment for arrays `vars` and `vals` passed to `SolutionGoal`. Thus `IloCplex` will call method `end` for these arrays as soon as they can be discarded. Calling `end` for the arrays passed to `SolutionGoal` is an error and must be avoided.

Inner Enumerations

Enumeration BranchType

Definition file: ilcplex/ilocplexi.h

`IloCplex::GoalI::BranchType` is an enumeration limited in scope to the class `IloCplex::GoalI`. This enumeration is used by the method `IloCplex::GoalI::getBranchType` to tell what kind of branch `IloCplex` is about to make:

- `BranchOnVariable` specifies branching on a single variable.
- `BranchOnAny` specifies multiple bound changes and constraints will be used for branching.
- `BranchOnSOS1` specifies branching on an SOS of type 1.
- `BranchOnSOS2` specifies branching on an SOS of type 2.

See Also: IloCplex::GoalI

Fields:

`BranchOnVariable = CPX_TYPE_VAR`	= CPX_TYPE_VAR
`BranchOnSOS1 = CPX_TYPE_SOS1`	= CPX_TYPE_SOS1
`BranchOnSOS2 = CPX_TYPE_SOS2`	= CPX_TYPE_SOS2
`BranchOnAny = CPX_TYPE_ANY`	= CPX_TYPE_ANY
`UserBranch = CPX_TYPE_USER`	

Enumeration IntegerFeasibility

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::GoalI::IntegerFeasibility` is an enumeration limited in scope to the class `IloCplex::GoalI`. This enumeration is used by `IloCplex::GoalI::getFeasibility` to access the integer feasibility of a variable or SOS in the current node solution:

- `Feasible` specifies the variable or SOS is integer feasible.
- `ImpliedFeasible` specifies the variable or SOS has been presolved out. It will be feasible when all other integer variables or SOS are integer feasible.
- `Infeasible` specifies the variable or SOS is integer infeasible.

See Also: IloCplex, GoalI::IntegerFeasibilityArray, ControlCallbackI::IntegerFeasibility

Fields:

`ImpliedInfeasible = -1`	
`Feasible = CPX_INTEGER_FEASIBLE`	Not applicable = CPX_INTEGER_FEASIBLE
`Infeasible = CPX_INTEGER_INFEASIBLE`	= CPX_INTEGER_INFEASIBLE
`ImpliedFeasible = CPX_IMPLIED_INTEGER_FEASIBLE`	= CPX_IMPLIED_INTEGER_FEASIBLE

Inner Typdefs

Typedef IntegerFeasibilityArray

Definition file: ilcplex/ilocplexi.h

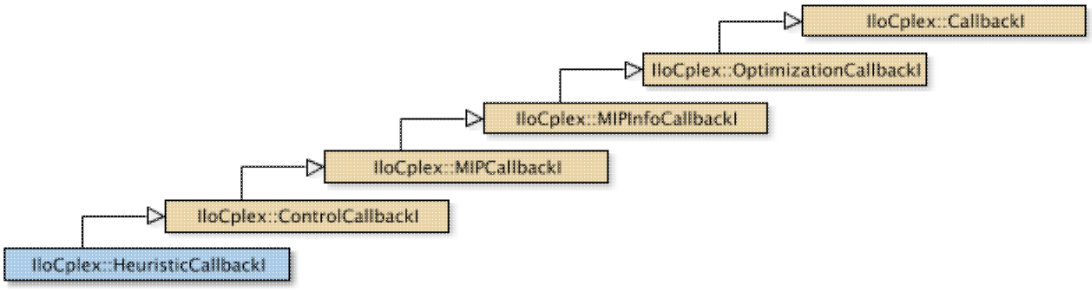
```
IloArray< IntegerFeasibility > IntegerFeasibilityArray
```

This type defines an array type for `IloCplex::GoalI::IntegerFeasibility`. The fully qualified name of an integer feasibility array is `IloCplex::GoalI::IntegerFeasibilityArray`.

See Also: IloCplex, GoalI::IntegerFeasibility

Class IloCplex::HeuristicCallbackI

Definition file: ilcplex/ilocplexi.h



Note

This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::HeuristicCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (MIP). When you derive a user-defined class of callbacks, this class offers protected methods for you to:

- give the instance of `IloCplex` a potential new incumbent solution;
- query the instance of `IloCplex` about the solution status for the current node;
- query the instance of `IloCplex` about the variable bounds at the current node;
- change bounds temporarily on a variable or group of variables at the current node;
- re-solve the problem at the node with the changed bounds;
- use all the query functions inherited from parent classes.

During branching, the heuristic callback is called after each node subproblem has been solved, including any cuts that may have been newly generated. Before branching, at the root node, the heuristic callback is also called before each round of cuts is added to the problem and re-solved.

In short, this callback allows you to attempt to construct an integer feasible solution at a node and pass it to the invoking instance of `IloCplex` to use as its new incumbent. The API supports you in finding such a solution by allowing you iteratively to change bounds of the variables and re-solve the node relaxation. Changing the bounds in the heuristic callback has no effect on the search beyond the termination of the callback.

If an attempt is made to access information not available at the node for the invoking instance of `IloCplex`, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::ControlCallbackI, IloCplex::MIPCallbackI, IloCplex::OptimizationCallbackI, ILOHEURISTICCALLBACK0

Method Summary	
`public IloCplex::CplexStatus`	`getCplexStatus() const`
`public IloAlgorithm::Status`	`getStatus() const`
`public IloBool`	`isDualFeasible() const`
`public IloBool`	`isPrimalFeasible() const`
`public void`	`setBounds(const IloIntVarArray var, const IloNumArray lb, const IloNumArray ub)`

public void	setBounds(const IloNumVarArray var, const IloNumArray lb, const IloNumArray ub)
public void	setBounds(const IloIntVar var, IloNum lb, IloNum ub)
public void	setBounds(const IloNumVar var, IloNum lb, IloNum ub)
public void	setSolution(const IloIntVarArray vars, const IloNumArray vals, IloNum obj)
public void	setSolution(const IloIntVarArray vars, const IloNumArray vals)
public void	setSolution(const IloNumVarArray vars, const IloNumArray vals, IloNum obj)
public void	setSolution(const IloNumVarArray vars, const IloNumArray vals)
public IloBool	solve(IloCplex::Algorithm alg=Dual)

Inherited Methods from **ControlCallbackI**
getDownPseudoCost, getDownPseudoCost, getFeasibilities, getFeasibilities, getFeasibility, getFeasibility, getFeasibility, getFeasibility, getLB, getLB, getLBs, getLBs, getNodeData, getObjValue, getSlack, getSlacks, getUB, getUB, getUBs, getUBs, getUpPseudoCost, getUpPseudoCost, getValue, getValue, getValue, getValues, getValues, isSOSFeasible, isSOSFeasible

Inherited Methods from **MIPCallbackI**
getNcliques, getNcovers, getNcuts, getNdisjunctiveCuts, getNflowCovers, getNflowPaths, getNfractionalCuts, getNGUBcovers, getNimpliedBounds, getNMIRs, getNzeroHalfCuts, getObjCoef, getObjCoef, getObjCoefs, getObjCoefs, getUserThreads

Inherited Methods from **MIPInfoCallbackI**
getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent

Inherited Methods from **OptimizationCallbackI**
getModel, getNcols, getNQCs, getNrows

Inherited Methods from **CallbackI**
abort, duplicateCallback, getEndTime, getEnv, main

Methods

public IloCplex::CplexStatus **getCplexStatus**() const

This method returns the CPLEX status of the instance of IloCplex at the current node (that is, the state of the optimizer at the node) during the last call to HeuristicCallbackI::solve (which may have been called directly in the callback or by IloCplex when processing the node).

The enumeration IloCplex::CplexStatus lists the possible status values.

public IloAlgorithm::Status **getStatus**() const

This method returns the status of the solution found by the instance of `IloCplex` at the current node during the last call to `HeuristicCallbackI::solve` (which may have been called directly in the callback or by `IloCplex` when processing the node).

The enumeration `IloAlgorithm::Status` lists the possible status values.

```
public IloBool isDualFeasible() const
```

This method returns `IloTrue` if the solution provided by the last `solve` call is dual feasible. Note that an `IloFalse` return value does not necessarily mean that the solution is not dual feasible. It simply means that the relevant algorithm was not able to conclude it was dual feasible when it terminated.

```
public IloBool isPrimalFeasible() const
```

This method returns `IloTrue` if the solution provided by the last `solve` call is primal feasible. Note that an `IloFalse` return value does not necessarily mean that the solution is not primal feasible. It simply means that the relevant algorithm was not able to conclude it was primal feasible when it terminated.

```
public void setBounds(const IloIntVarArray var, const IloNumArray lb, const
IloNumArray ub)
```

For each variable in the array `var`, this method sets its upper bound to the corresponding value in the array `ub` and its lower bound to the corresponding value in the array `lb`, provided `var` has not been removed by presolve. Setting bounds has no effect beyond the scope of the current invocation of the callback.

When using this method, you must avoid changing the bounds of a variable that has been removed by presolve. To check whether presolve is off, consider the parameter `IloCplex::PreInd`. Alternatively, you can check whether a particular variable has been removed by presolve by checking the status of the variable. To do so, call `IloCplex::ControlCallback::getFeasibilities`. A variable that has been removed by presolve will have the status `ImpliedFeasible`.

```
public void setBounds(const IloNumVarArray var, const IloNumArray lb, const
IloNumArray ub)
```

For each variable in the array `var`, this method sets its upper bound to the corresponding value in the array `ub` and its lower bound to the corresponding value in the array `lb`, provided the variable has not been removed by presolve. Setting bounds has no effect beyond the scope of the current invocation of the callback.

```
public void setBounds(const IloIntVar var, IloNum lb, IloNum ub)
```

This method sets the lower bound to `lb` and the upper bound to `ub` for the variable `var` at the current node, provided `var` has not been removed by presolve. Setting bounds has no effect beyond the scope of the current invocation of the callback.

When using this method, you must avoid changing the bounds of a variable that has been removed by presolve. To check whether presolve is off, consider the parameter `IloCplex::PreInd`. Alternatively, you can check whether a particular variable has been removed by presolve by checking the status of the variable. To do so, call `IloCplex::ControlCallback::getFeasibilities`. A variable that has been removed by presolve will have the status `ImpliedFeasible`.

```
public void setBounds(const IloNumVar var, IloNum lb, IloNum ub)
```

This method sets the lower bound to `lb` and the upper bound to `ub` for the variable `var` at the current node, provided `var` has not been removed by presolve. Setting bounds has no effect beyond the scope of the current invocation of the callback.

When using this method, you must avoid changing the bounds of a variable that has been removed by presolve. To check whether presolve is off, consider the parameter `IloCplex::PreInd`. Alternatively, you can check whether a particular variable has been removed by presolve by checking the status of the variable. To do so, call `IloCplex::ControlCallback::getFeasibilities`. A variable that has been removed by presolve will have the status `ImpliedFeasible`.

```
public void setSolution(const IloIntVarArray vars, const IloNumArray vals, IloNum
obj)
```

For each variable in the array `vars`, this method uses the value in the corresponding element of the array `vals` to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, `setSolution` can be used to pass it over to `IloCplex`. `IloCplex` then analyses the solution and, if it is both feasible and better than the current incumbent, uses it as the new incumbent. A solution is specified using arrays `vars` and `vals`, where `vals[i]` specifies the solution value for `vars[i]`.

The parameter `obj` is used to tell `IloCplex` the objective value of the injected solution. This allows `IloCplex` to skip the computation of that value, but care must be taken not to provide an incorrect value.

Do not call this method multiple times. Calling it again will overwrite any previously specified solution.

```
public void setSolution(const IloIntVarArray vars, const IloNumArray vals)
```

For each variable in the array `vars`, this method uses the value in the corresponding element of the array `vals` to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, `setSolution` can be used to pass it over to `IloCplex`. `IloCplex` then analyses the solution and, if it is both feasible and better than the current incumbent, uses it as the new incumbent. A solution is specified using arrays `vars` and `vals`, where `vals[i]` specifies the solution value for `vars[i]`.

Do not call this method multiple times. Calling it again will overwrite any previously specified solution.

```
public void setSolution(const IloNumVarArray vars, const IloNumArray vals, IloNum
obj)
```

For each variable in the array `vars`, this method uses the value in the corresponding element of the array `vals` to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, `setSolution` can be used to pass it over to `IloCplex`. `IloCplex` then analyses the solution and, if it is both feasible and better than the current incumbent, uses it as the new incumbent. A solution is specified using arrays `vars` and `vals`, where `vals[i]` specifies the solution value for `vars[i]`.

The parameter `obj` is used to tell `IloCplex` the objective value of the injected solution. This allows `IloCplex` to skip the computation of that value, but care must be taken not to provide an incorrect value.

Do not call this method multiple times. Calling it again will overwrite any previously specified solution.

```
public void setSolution(const IloNumVarArray vars, const IloNumArray vals)
```

For each variable in the array `vars`, this method uses the value in the corresponding element of the array `vals` to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, `setSolution` can be used to pass it over to `IloCplex`. `IloCplex` then analyses the solution and, if it is both feasible and better than the current incumbent, `IloCplex` uses it as the new incumbent. A solution is specified using arrays `vars` and `vals`, where `vals[i]` specifies the solution value for `vars[i]`.

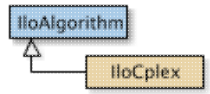
Do not call this method multiple times. Calling it again will overwrite any previously specified solution.

```
public IloBool solve(IloCplex::Algorithm alg=Dual)
```

This method can be used to solve the current node relaxation, usually after some bounds have been changed by `HeuristicCallbackI::setBounds`. By default it uses the dual simplex algorithm, but this behavior can be overridden by the optional parameter `alg`. See the enumeration `IloCplex::Algorithm` for a list of the available optimizers.

Class IloAlgorithm

Definition file: ilconcert/iloalg.h



The base class of algorithms in Concert Technology.
An instance of `IloAlgorithm` represents an algorithm in Concert Technology.

In general terms, you define a model, and Concert Technology extracts objects from it for your target algorithm and then solves for solutions.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Status

The member function `getStatus` returns a status showing information about the currently extracted model and the solution (if there is one). For explanations of the status, see the nested enumeration `IloAlgorithm::Status`.

Exceptions

The class `IloAlgorithm::Exception`, derived from the class `IloException`, is the base class of exceptions thrown by classes derived from `IloAlgorithm`. For an explanation of exceptions thrown by instances of `IloAlgorithm`, see `IloAlgorithm::Exception`.

Streams and Output

The class `IloAlgorithm` supports these communication streams:

- `ostream& IloAlgorithm::error() const;` for error messages.
- `ostream& IloAlgorithm::out() const;` for general output.
- `ostream& IloAlgorithm::warning() const;` for warning messages about nonfatal conditions.

Child classes:

- the class `IloCplex` in the *IBM ILOG CPLEX Reference Manual*
- the class `IloCP` in the *IBM ILOG CP Optimizer Reference Manual*.
- the class `IloSolver` in the *IBM ILOG Solver Reference Manual* .

See Also: IloEnv, IloModel, IloAlgorithm::Status, IloAlgorithm::Exception

Constructor Summary	
public	IloAlgorithm(IloAlgorithmI * impl=0)

Method Summary	
public void	clear() const
public void	end()
public ostream &	error() const
public void	extract(const IloModel) const
public IloEnv	getEnv() const
public IloInt	getIntValue(const IloIntVar) const

public void	getIntValues(const IloIntVarArray, IloIntArray) const
public IloModel	getModel() const
public IloNum	getObjValue() const
public IloAlgorithm::Status	getStatus() const
public IloNum	getTime() const
public IloNum	getValue(const IloNumExprArg) const
public IloNum	getValue(const IloObjective) const
public IloNum	getValue(const IloIntVar) const
public IloNum	getValue(const IloNumVar) const
public void	getValues(const IloIntVarArray, IloNumArray) const
public void	getValues(const IloNumVarArray, IloNumArray) const
public IloBool	isExtracted(const IloExtractable) const
public ostream &	out() const
public void	printTime() const
public void	resetTime() const
public void	setError(ostream &)
public void	setOut(ostream &)
public void	setWarning(ostream &)
public IloBool	solve() const
public ostream &	warning() const

Inner Enumeration	
IloAlgorithm::Status	An enumeration for the class `IloAlgorithm`.

Inner Class	
IloAlgorithm::CannotExtractException	The class of exceptions thrown if an object cannot be extracted from a model.
IloAlgorithm::CannotRemoveException	The class of exceptions thrown if an object cannot be removed from a model.
IloAlgorithm::Exception	The base class of exceptions thrown by classes derived from IloAlgorithm.
IloAlgorithm::NotExtractedException	The class of exceptions thrown if an extractable object has no value in the current solution of an algorithm.

Constructors

```
public IloAlgorithm(IloAlgorithmI * impl=0)
```

This constructor creates an algorithm in Concert Technology from its implementation object. This is the default constructor.

Methods

```
public void clear() const
```

This member function clears the current model from the algorithm.

```
public void end()
```

This member function deletes the invoking algorithm. That is, it frees memory associated with the invoking algorithm.

```
public ostream & error() const
```

This member function returns a reference to the stream currently used for error messages from the invoking algorithm. `IloAlgorithm::error` is initialized with the value of `IloEnv::error`.

```
public void extract(const IloModel) const
```

This member function extracts the extractable objects from a model into the invoking algorithm if a member function exists to extract the objects from the model for the invoking algorithm. Not all extractable objects can be extracted by all algorithms; see the documentation of the algorithm class you are using for a list of extractable classes it supports.

When you use this member function to extract extractable objects from a model, it extracts all the elements of that model for which Concert Technology creates the representation of the extractable object suitable for the invoking algorithm.

The attempt to extract may fail. In case such a failure occurs, Concert Technology throws the exception `CannotExtractException` on platforms that support C++ exceptions when exceptions are enabled.

For example, a failure will occur if you attempt to extract more than one objective for an invoking algorithm that accepts only one objective, and Concert Technology will throw the exception `MultipleObjException`.

```
public IloEnv getEnv() const
```

This member function returns the environment of the invoking algorithm.

```
public IloInt getIntValue(const IloIntVar) const
```

This member function returns the integer value of an integer variable in the current solution of the invoking algorithm. For example, to access the variable, use the member function `getIntValue(var)` where `var` is an instance of the class `IloIntVar`.

If there is no value to return for `var`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `var` was not extracted by the invoking algorithm).

```
public void getIntValues(const IloIntVarArray, IloIntArray) const
```

This member function accepts an array of variables `vars` and puts the corresponding values into the array `vals`; the corresponding values come from the current solution of the invoking algorithm. The array `vals` must be a clean, empty array when you pass it to this member function.

If there are no values to return for `vars`, this member function raises an error. On platforms that support C++ exceptions, when exceptions are enabled, this member function throws the exception `NotExtractedException` in such a case.

```
public IloModel getModel() const
```

This member function returns the model of the invoking algorithm.

```
public IloNum getObjValue() const
```

This member function returns the numeric value of the objective function associated with the invoking algorithm.

```
public IloAlgorithm::Status getStatus() const
```

This member function returns a status showing information about the current model and the solution. For explanations of the status, see the nested enumeration `IloAlgorithm::Status`.

```
public IloNum getTime() const
```

This member function returns the amount of time elapsed in seconds since the most recent reset of the invoking algorithm. (The member function `IloAlgorithm::printTime` directs the output of `getTime` to the output channel of the invoking algorithm.)

See Also: IloTimer

```
public IloNum getValue(const IloNumExprArg) const
```

This member function returns the value of an expression in the current solution of the invoking algorithm. For example, to access the expression, use the member function `getValue(expr)` where `expr` is an instance of the class `IloNumExprArg`.

If there is no value to return for `expr`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `expr` was not extracted by the invoking algorithm).

```
public IloNum getValue(const IloObjective) const
```

This member function returns the value of an objective in the current solution of the invoking algorithm. For example, to access the objective, use the member function `getValue(obj)` where `obj` is an instance of the class `IloObjective`.

If there is no value to return for `obj`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `obj` was not extracted by the invoking algorithm).

```
public IloNum getValue(const IloIntVar) const
```

This member function returns the numeric value of an integer variable in the current solution of the invoking algorithm. For example, to access the variable, use the member function `getValue(var)` where `var` is an instance of the class `IloIntVar`.

If there is no value to return for `var`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `var` was not extracted by the invoking algorithm).

```
public IloNum getValue(const IloNumVar) const
```

This member function returns the numeric value of a numeric variable in the current solution of the invoking algorithm. For example, to access the value of the variable, use the member function `getValue(var)` where `var` is an instance of the class `IloNumVar`.

If there is no value to return for `var`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `var` was not extracted by the invoking algorithm).

```
public void getValues(const IloIntVarArray, IloNumArray) const
```

This member function accepts an array of variables `vars` and puts the corresponding values into the array `vals`; the corresponding values come from the current solution of the invoking algorithm. The array `vals` must be a clean, empty array when you pass it to this member function.

If there are no values to return for `vars`, this member function raises an error. On platforms that support C++ exceptions, when exceptions are enabled, this member function throws the exception `NotExtractedException` in such a case.

```
public void getValues(const IloNumVarArray, IloNumArray) const
```

This member function accepts an array of variables `vars` and puts the corresponding values into the array `vals`; the corresponding values come from the current solution of the invoking algorithm. The array `vals` must be a clean, empty array when you pass it to this member function.

If there are no values to return for `vars`, this member function raises an error. On platforms that support C++ exceptions, when exceptions are enabled, this member function throws the exception `NotExtractedException` in such a case.

```
public IloBool isExtracted(const IloExtractable) const
```

This member function returns `IloTrue` if `extr` has been extracted for the invoking algorithm; otherwise, it returns `IloFalse`.

```
public ostream & out() const
```

This member function returns a reference to the stream currently used for logging. General output from the invoking algorithm is accessible through this member function. `IloAlgorithm::out` is initialized with the value of `IloEnv::out`.

```
public void printTime() const
```

This member function directs the output of the member function `IloAlgorithm::getTime` to an output channel of the invoking algorithm. (The member function `IloAlgorithm::getTime` accesses the elapsed time in seconds since the most recent reset of the invoking algorithm.)

```
public void resetTime() const
```

This member function resets the timer on the invoking algorithm. The type of timer is platform dependent. On Windows systems, the time is elapsed wall clock time. On UNIX systems, the time is CPU time.

```
public void setError(ostream &)
```

This member function sets the stream for errors generated by the invoking algorithm. By default, the stream is defined by an instance of `IloEnv` as `cerr`.

```
public void setOut(ostream &)
```

This member function redirects the `out()` stream with the stream given as an argument.

This member function can be used with `IloEnv::getNullStream` to suppress screen output by redirecting it to the null stream.

```
public void setWarning(ostream &)
```

This member function sets the stream for warnings from the invoking algorithm. By default, the stream is defined by an instance of `IloEnv` as `cout`.

```
public IloBool solve() const
```

This member function solves the current model in the invoking algorithm. In other words, `solve` works with all extractable objects extracted from the model for the algorithm. The member function returns `IloTrue` if it finds a solution (not necessarily an optimal one). Here is an example of its use:

```
if (algo.solve()) {
  algo.out() << "Status is " << algo.getStatus() << endl;
};
```

If an objective of the model has been extracted into the invoking algorithm, this member function solves the model to optimality. If there is currently no objective, this member function searches for the first feasible solution. A feasible solution is not necessarily optimal, though it satisfies all constraints.

```
public ostream & warning() const
```

This member function returns a reference to the stream currently used for warnings from the invoking algorithm. `IloAlgorithm::warning` is initialized with the value of `IloEnv::warning`.

Inner Enumerations

Enumeration Status

Definition file: ilconcert/iloalg.h

An enumeration for the class `IloAlgorithm`.
`IloAlgorithm` is the base class of algorithms in Concert Technology, and `IloAlgorithm::Status` is an enumeration limited in scope to the class `IloAlgorithm`. The member function `IloAlgorithm::getStatus` returns a status showing information about the current model and the solution.

`Unknown` specifies that the algorithm has no information about the solution of the model.

`Feasible` specifies that the algorithm found a feasible solution (that is, an assignment of values to variables that satisfies the constraints of the model, though it may not necessarily be optimal). The member functions `IloAlgorithm::getValue` access this feasible solution.

`Optimal` specifies that the algorithm found an optimal solution (that is, an assignment of values to variables that satisfies all the constraints of the model and that is proved optimal with respect to the objective of the model). The member functions `IloAlgorithm::getValue` access this optimal solution.

`Infeasible` specifies that the algorithm proved the model infeasible; that is, it is not possible to find an assignment of values to variables satisfying all the constraints in the model.

`Unbounded` specifies that the algorithm proved the model unbounded.

`InfeasibleOrUnbounded` specifies that the model is infeasible or unbounded.

`Error` specifies that an error occurred and, on platforms that support exceptions, that an exception has been thrown.

See Also: IloAlgorithm, operator<<

Fields:

`Unknown`

`Feasible`

`Optimal`

`Infeasible`

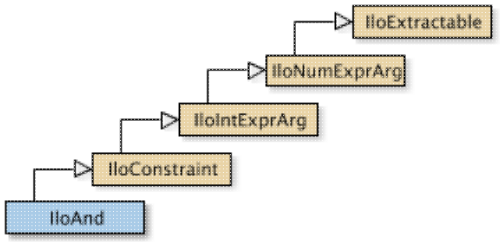
`Unbounded`

`InfeasibleOrUnbounded`

`Error`

Class IloAnd

Definition file: ilconcert/ilomodel.h



Defines a logical conjunctive-AND among other constraints.
An instance of `IloAnd` represents a conjunctive constraint. In other words, it defines a logical conjunctive-AND among any number of constraints. It lets you represent a constraint on constraints in your model. Since an instance of `IloAnd` is a constraint itself, you can build up extensive conjunctions by adding constraints to an instance of `IloAnd` by means of the member function `IloAnd::add`. You can also remove constraints from an instance of `IloAnd` by means of the member function `IloAnd::remove`.

The elements of a conjunctive constraint must be in the same environment.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Conjunction of Goals

If you want to represent the conjunction of goals (rather than constraints) in your model, then you should consider the function `IloAndGoal` (documented in the IBM ILOG Solver Reference Manual).

What Is Extracted

All the constraints (that is, instances of `IloConstraint` or one of its subclasses) that have been added to a conjunctive constraint (an instance of `IloAnd`) and that have not been removed from it will be extracted when an algorithm such as `IloCplex`, `IloCP`, or `IloSolver` extracts the constraint.

Example

For example, you may write:

```
IloAnd and(env);
and.add(constraint1);
and.add(constraint2);
and.add(constraint3);
```

Those lines are equivalent to :

```
IloAnd and = constraint1 && constraint2 && constraint3;
```

See Also: IloConstraint, IloOr, operator&&

Constructor Summary	
`public`	`IloAnd()`

public	IloAnd(IloAndI * impl)
public	IloAnd(const IloEnv env, const char * name=0)

Method Summary	
public void	add(const IloConstraintArray array) const
public void	add(const IloConstraint constraint) const
public IloAndI *	getImpl() const
public void	remove(const IloConstraintArray array) const
public void	remove(const IloConstraint constraint) const

Inherited Methods from `IloConstraint`
getImpl

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

`public` **`IloAnd`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloAnd`**`(IloAndI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IloAnd`**`(const IloEnv env, const char * name=0)`

This constructor creates a conjunctive constraint for use in the environment `env`. In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

The optional argument `name` is set to `0` by default.

Methods

`public void` **`add`**`(const IloConstraintArray array) const`

This member function makes all the elements in `array` elements of the invoking conjunctive constraint. In other words, it applies the invoking conjunctive constraint to all the elements of `array`.

Note

The member function `add` notifies Concert Technology algorithms about this change to the invoking object.

```
public void add(const IloConstraint constraint) const
```

This member function makes `constraint` one of the elements of the invoking conjunctive constraint. In other words, it applies the invoking conjunctive constraint to `constraint`.

Note

The member function `add` notifies Concert Technology algorithms about this change to the invoking object.

```
public IloAndI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void remove(const IloConstraintArray array) const
```

This member function removes all the elements of `array` from the invoking conjunctive constraint so that the invoking conjunctive constraint no longer applies to any of those elements.

Note

The member function `remove` notifies Concert Technology algorithms about this change to the invoking object.

```
public void remove(const IloConstraint constraint) const
```

This member function removes `constraint` from the invoking conjunctive constraint so that the invoking conjunctive constraint no longer applies to `constraint`.

Note

The member function `remove` notifies Concert Technology algorithms about this change to the invoking object.

Class IloArray<>

Definition file: ilconcert/iloenv.h

A template to create classes of arrays for elements of a given class.
This C++ template creates a class of arrays for elements of a given class. In other words, you can use this template to create arrays of Concert Technology objects; you can also use this template to create arrays of arrays (that is, multidimensional arrays).

In its synopsis, `X` represents a class, `x` is an instance of the class `X`. This template creates the array class (`IloArrayX`) for any class in Concert Technology, including classes with names in the form `IloXArray`, such as `IloExtractableArray`. Concert Technology predefines the array classes listed here as **See Also**. The member functions defined by this template are documented in each of those predefined classes.

The classes you create in this way consist of extensible arrays. That is, you can add elements to the array as needed.

Deleting Arrays

The member function `end` created by this template deletes only the array; the member function does not delete the elements of the array.

Copying Arrays

Like certain other Concert Technology classes, a class of arrays created by `IloArray` is a handle class corresponding to an implementation class. In other words, an instance of an `IloArray` class is a handle pointing to a corresponding implementation object. More than one handle may point to the same implementation object.

Input and Output of Multidimensional Arrays

The template `operator >>` makes it possible to read numeric values from a file in the format `[x, y, z, ...]` where `x`, `y`, `z` are the results of the `operator >>` for class `X`. Class `X` must provide a default constructor for `operator >>` to work. That is, the statement `X x;` must work for `X`. This input operator is limited to numeric values.

Likewise, the template `operator <<` makes it possible to write to a file in the format `[x, y, z, ...]` where `x`, `y`, `z` are the results of the `operator <<` for class `X`. (This output operator is *not* limited to numeric values, as the input operator is.)

These two operators make it possible to read and write multidimensional arrays of numeric values like this:

```
IloArray<IloArray<IloIntArray> >
```

(Notice the space between > > at the end of that statement. It is necessary in C++.)

However, there is a practical limit of four on the number of dimensions supported by the input operator for reading multidimensional arrays. This limit is due to the inability of certain C++ compilers to support templates correctly. Specifically, you can read input by means of the input operator for multidimensional arrays of one, two, three, or four dimensions. There is no such limit on the number of dimensions with respect to the output operator for multidimensional arrays.

See Also these classes in the *IBM ILOG CPLEX Reference Manual*: `IloSemiContVarArray`, `IloSOS1Array`, `IloSOS2Array`, `IloNumColumnArray`.

See Also these classes in the *IBM ILOG Solver Reference Manual*: `IloAnyArray`, `IloAnySetVarArray`, `IloAnyVarArray`, `IloFloatArray`, `IloFloatVarArray`.

See Also: IloBoolArray, IloBoolVarArray, IloConstraintArray, IloExprArray, IloExtractableArray, IloIntArray, IloIntVarArray, IloNumVarArray, IloRangeArray, IloSolutionArray

Constructor Summary	
public	IloArray(IloEnv env, IloInt max=0)

Method Summary	
public void	add(IloArray< X > ax) const
public void	add(IloInt more, X x) const
public void	add(X x) const
public void	clear()
public void	end()
public IloEnv	getEnv() const
public IloInt	getSize() const
public X &	operator[](IloInt i)
public const X &	operator[](IloInt i) const
public void	remove(IloInt first, IloInt nb=1)

Constructors

`public` **`IloArray`**`(IloEnv env, IloInt max=0)`

This constructor creates an array of `max` elements, all of which are empty handles.

Methods

`public void` **`add`**`(IloArray< X > ax) const`

This member function appends the elements in `ax` to the invoking array.

`public void` **`add`**`(IloInt more, X x) const`

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

`public void` **`add`**`(X x) const`

This member function appends `x` to the invoking array.

`public void` **`clear`**`()`

This member function removes all the elements from the invoking array. In other words, it produces an empty array.

```
public void end()
```

This member function first removes the invoking extractable object from all other extractable objects where it is used (such as a model, ranges, etc.) and then deletes the invoking extractable object. That is, it frees all the resources used by the invoking object. After a call to this member function, you cannot use the invoking extractable object again.

```
public IloEnv getEnv() const
```

This member function returns the environment where the invoking array was created. The elements of the invoking array belong to the same environment.

```
public IloInt getSize() const
```

This member function returns an integer specifying the size of the invoking array. An empty array has size 0 (zero).

```
public X & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`.

```
public const X & operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

```
 IloArray operator[] (IloInt i) const;
```

```
public void remove(IloInt first, IloInt nb=1)
```

This member function removes elements from the invoking array. It begins removing elements at the index specified by `first`, and it removes `nb` elements (`nb = 1` by default).

Class IloBarrier

Definition file: ilconcert/ilothread.h

A system class to synchronize threads at a specified number.
The class `IloBarrier` provides synchronization primitives adapted to Concert Technology. A barrier, an instance of this class, serves as a rendezvous for a specific number of threads. After you create a barrier for n threads, the first n-1 threads to reach that barrier will be blocked. The nth thread to arrive at the barrier completes the synchronization and wakes up the n-1 threads already waiting at that barrier. When the nth thread arrives, the barrier resets itself. Any other thread that arrives at this point is blocked and will participate in a new barrier of size n.

Note

The class `IloBarrier` has nothing to do with the IBM ILOG CPLEX barrier optimizer.

System Class

`IloBarrier` is a system class.

Most Concert Technology classes are actually handle classes whose instances point to objects of a corresponding implementation class. For example, instances of the Concert Technology class `IloNumVar` are handles pointing to instances of the implementation class `IloNumVarI`. Their allocation and de-allocation in a Concert Technology environment are managed by an instance of `IloEnv`.

However, system classes, such as `IloBarrier`, differ from that Concert Technology pattern. `IloBarrier` is an ordinary C++ class. Its instances are allocated on the C++ heap.

Instances of `IloBarrier` are not automatically de-allocated by a call to `IloEnv::end`. You must explicitly destroy instances of `IloBarrier` by means of a call to the delete operator (which calls the appropriate destructor) when your application no longer needs instances of this class.

Furthermore, you should not allocate—neither directly nor indirectly—any instance of `IloBarrier` in a Concert Technology environment because the destructor for that instance of `IloBarrier` will never be called automatically by `IloEnv::end` when it cleans up other Concert Technology objects in that Concert Technology environment.

For example, it is not a good idea to make an instance of `IloBarrier` part of a conventional Concert Technology model allocated in a Concert Technology environment because that instance will not automatically be de-allocated from the Concert Technology environment along with the other Concert Technology objects.

De-allocating Instances of IloBarrier

Instances of `IloBarrier` differ from the usual Concert Technology objects because they are not allocated in a Concert Technology environment, and their de-allocation is not managed automatically for you by `IloEnv::end`. Instead, you must explicitly destroy instances of `IloBarrier` by calling the delete operator when your application no longer needs those objects.

See Also: IloCondition, IloFastMutex

Constructor Summary	
`public`	`IloBarrier(int count)`

Method Summary

public int	wait()

Constructors

public **IloBarrier**(int count)

This constructor creates an instance of IloBarrier of size count and allocates it on the C++ heap (not in a Concert Technology environment).

Methods

public int **wait**()

The first count-1 calls to this member function block the calling thread. The last call (that is, the call numbered count) wakes up all the count-1 waiting threads. Once a thread has been woken up, it leaves the barrier. When a thread leaves the barrier (that is, when it returns from the wait call), it will return either 1 (one) or 0 (zero). If the thread returns 0, the barrier is not yet empty. If the thread returns 1, it was the last thread at the barrier.

A nonempty barrier contains blocked threads or exiting threads.

Class IloBaseEnvMutex

Definition file: ilconcert/iloenv.h

A class to initialize multithreading in an application.
An instance of this base class in the function `IloInitMT` initializes multithreading in a Concert Technology application. For a general purpose mutex, see the class `IloFastMutex`.

See Also: IloFastMutex, IloInitMT

Method Summary	
`public virtual void`	`lock()`
`public virtual void`	`unlock()`

Methods

`public virtual void` **`lock`**`()`

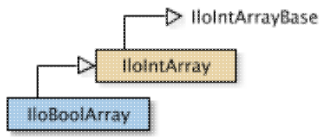
This member function locks a mutex.

`public virtual void` **`unlock`**`()`

This member function unlocks a mutex.

Class IloBoolArray

Definition file: ilconcert/iloenv.h



The array class of the basic Boolean class for a model.
`IloBoolArray` is the array class of the basic Boolean class for a model. It is a handle class. The implementation class for `IloBoolArray` is the undocumented class `IloBoolArrayI`.

Instances of `IloBoolArray` are extensible. (They differ from instances of `IlcBoolArray` in this respect.) References to an array change whenever an element is added to or removed from the array.

For each basic type, Concert Technology defines a corresponding array class. That array class is a handle class. In other words, an object of that class contains a pointer to another object allocated in a Concert Technology environment associated with a model. Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object: as a developer using handles, you do not have to worry about memory allocation.

As handles, these objects should be passed by value, and they should be created as automatic objects, where "automatic" has the usual C++ meaning.

Member functions of a handle class correspond to member functions of the same name in the implementation class.

Assert and NDEBUG

Most member functions of the class `IloBoolArray` are inline functions that contain an `assert` statement. This statement checks that the handle pointer is not null. These statements can be suppressed by the macro `NDEBUG`. This option usually reduces execution time. The price you pay for this choice is that attempts to access through null pointers are not trapped and usually result in memory faults.

See Also: IloBool

Constructor Summary	
public	`IloBoolArray(IloArrayI * i=0)`
public	`IloBoolArray(const IloEnv env, IloInt n=0)`
public	`IloBoolArray(const IloEnv env, IloInt n, const IloBool v0, const IloBool v1...)`

Method Summary	
public void	`add(IloInt more, const IloBool x)`
public void	`add(const IloBool x)`
public void	`add(const IloBoolArray x)`

Inherited Methods from `IloIntArray`
`contains, contains, discard, discard, operator[], operator[], operator[], toNumArray`

Constructors

```
public IloBoolArray(IloArrayI * i=0)
```

This constructor creates an array of Boolean values from an implementation object.

```
public IloBoolArray(const IloEnv env, IloInt n=0)
```

This constructor creates an array of `n` Boolean values for use in a model in the environment specified by `env`. By default, its elements are empty handles.

```
public IloBoolArray(const IloEnv env, IloInt n, const IloBool v0, const IloBool
v1...)
```

This constructor creates an array of `n` Boolean values; the elements of the new array take the corresponding values: `v0, v1, ...,v(n-1)`.

Methods

```
public void add(IloInt more, const IloBool x)
```

This member function appends `x` to the invoking array of Boolean values; it appends `x more` times.

```
public void add(const IloBool x)
```

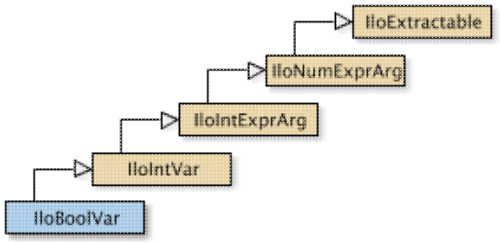
This member function appends the value `x` to the invoking array.

```
public void add(const IloBoolArray x)
```

This member function appends the values in the array `x` to the invoking array.

Class IloBoolVar

Definition file: ilconcert/iloexpression.h



An instance of this class represents a constrained Boolean variable in a Concert Technology model.
Boolean variables are also known as binary decision variables. They can assume the values 0 (zero) or 1 (one).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

What Is Extracted

An instance of `IloBoolVar` is extracted by `IloSolver` (documented in the *IBM ILOG Solver Reference Manual*) as an instance of the class `IlcBoolVar` (also documented in the *IBM ILOG Solver Reference Manual).*

An instance of `IloBoolVar` is extracted by `IloCplex` (documented in the *IBM ILOG CPLEX Reference Manual)* as a column representing a numeric variable of type `Bool` with bounds as specified by `IloBoolVar`.

See Also: IloIntVar, IloNumVar

Constructor Summary	
`public`	`IloBoolVar(IloEnv env, IloInt min=0, IloInt max=1, const char * name=0)`
`public`	`IloBoolVar(IloEnv env, const char * name)`
`public`	`IloBoolVar(const IloAddNumVar & column, const char * name=0)`

Inherited Methods from `IloIntVar`
`getImpl, getLB, getMax, getMin, getUB, setBounds, setLB, setMax, setMin, setPossibleValues, setUB`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

```
public IloBoolVar(IloEnv env, IloInt min=0, IloInt max=1, const char * name=0)
```

This constructor creates a Boolean variable and makes it part of the environment `env`. By default, the Boolean variable assumes a value of 0 (zero) or 1 (one). By default, its name is the empty string, but you can specify a name of your own choice.

```
public IloBoolVar(IloEnv env, const char * name)
```

This constructor creates a Boolean variable and makes it part of the environment `env`. By default, its name is the empty string, but you can specify a name of your own choice.

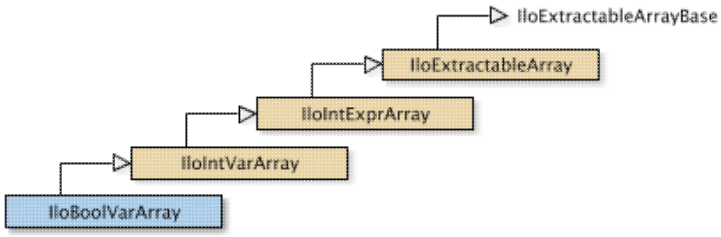
```
public IloBoolVar(const IloAddNumVar & column, const char * name=0)
```

This constructor creates an instance of `IloBoolVar` like this:

```
IloNumVar(column, 0.0, 1.0, ILOBOOL, name);
```

Class IloBoolVarArray

Definition file: ilconcert/iloexpression.h



The array class of the Boolean variable class.
For each basic type, Concert Technology defines a corresponding array class. `IloBoolVarArray` is the array class of the Boolean variable class for a model. It is a handle class.

Instances of `IloBoolVarArray` are extensible.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloBoolVar

Constructor Summary	
public	IloBoolVarArray(IloDefaultArrayI * i=0)
public	IloBoolVarArray(const IloEnv env, IloInt n)

Method Summary	
public void	add(IloInt more, const IloBoolVar x)
public void	add(const IloBoolVar x)
public void	add(const IloBoolVarArray x)
public IloBoolVar	operator[](IloInt i) const
public IloBoolVar &	operator[](IloInt i)
public IloIntExprArg	operator[](IloIntExprArg anIntegerExpr) const

Inherited Methods from `IloIntVarArray`
add, add, add, endElements, operator[], operator[], operator[], toNumVarArray

Inherited Methods from `IloIntExprArray`
add, add, add, endElements, operator[], operator[], operator[]

Inherited Methods from `IloExtractableArray`
add, add, add, endElements, setNames

Constructors

`public` **`IloBoolVarArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty extensible array of Boolean variables.

```
public IloBoolVarArray(const IloEnv env, IloInt n)
```

This constructor creates an extensible array of `n` Boolean variables.

Methods

```
public void add(IloInt more, const IloBoolVar x)
```

This member function appends `x` to the invoking array of Boolean variables. The argument `more` specifies how many times.

```
public void add(const IloBoolVar x)
```

This member function appends the value `x` to the invoking array.

```
public void add(const IloBoolVarArray x)
```

This member function appends the variables in the array `x` to the invoking array.

```
public IloBoolVar operator[](IloInt i) const
```

This operator returns a reference to the extractable object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

```
IloBoolVar operator[] (IloInt i) const;
```

```
public IloBoolVar & operator[](IloInt i)
```

This operator returns a reference to the extractable object located in the invoking array at the position specified by the index `i`.

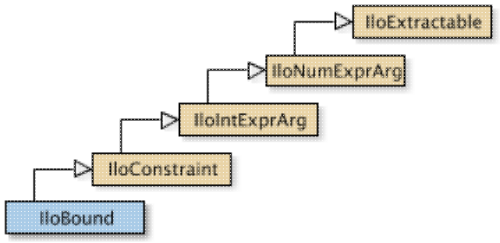
```
public IloIntExprArg operator[](IloIntExprArg anIntegerExpr) const
```

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element expression.

Class IloBound

Definition file: ilcplex/ilocplexi.h



This class represents a bound as a constraint in a conflict.

Constructor Summary	
public	IloBound()
public	IloBound(IloBoundI * impl)
public	IloBound(IloNumVar var, IloBound::Type type)

Method Summary	
public IloBoundI *	getImpl() const
public IloBound::Type	getType()
public IloNumVar	getVar()

Inherited Methods from `IloConstraint`
getImpl

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Inner Enumeration
IloBound::Type

Constructors

`public` **`IloBound`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloBound`**`(IloBoundI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

```
public IloBound(IloNumVar var, IloBound::Type type)
```

This constructor creates a bound for use in conflicts.

Methods

```
public IloBoundI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloBound::Type getType()
```

Accesses the bound specified by the invoking object.

```
public IloNumVar getVar()
```

Accesses the variable for which the invoking object specifies a bound.

Inner Enumerations

Enumeration Type

Definition file: ilcplex/ilocplexi.h

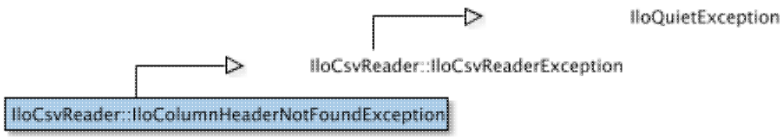
This enumeration lists the types of bounds that may appear in a conflict.

Fields:

```
Lower
Upper
```

Class IloCsvReader::IloColumnHeaderNotFoundException

Definition file: ilconcert/ilocsvreader.h



Exception thrown for unfound header.
This exception is thrown by the member functions listed below if a header (column name) that you use does not exist.

- IloCsvLine::getFloatByHeader
- IloCsvLine::getIntByHeader
- IloCsvLine::getStringByHeader
- IloCsvLine::getFloatByHeaderOrDefaultValue
- IloCsvLine::getIntByHeaderOrDefaultValue
- IloCsvLine::getStringByHeaderOrDefaultValue
- IloCsvReader::getPosition
- IloCsvTableReader::getPosition

Class IloCondition

Definition file: ilconcert/ilothread.h

Provides synchronization primitives adapted to Concert Technology for use in a parallel application.
The class `IloCondition` provides synchronization primitives adapted to Concert Technology for use in a parallel application.

An instance of the class `IloCondition` allows several threads to synchronize on a specific event. In this context, inter-thread communication takes place through signals. A thread expecting a condition of the computation state (say, `conditionC`) to be true before it executes a `treatmentT` can wait until the condition is true. When computation reaches a state where `conditionC` holds, then another thread can signal this fact by notifying a single waiting thread or by broadcasting to all the waiting threads that `conditionC` has now been met.

The conventional template for waiting on `conditionC` looks like this:

```
mutex.lock();
while (conditionC does not hold)
       condition.wait(&mutex);
doTreatmentT();
mutex.unlock();
```

That template has the following properties:

- The whole fragment is a critical section so that the evaluation of `conditionC` is protected. (Indeed, it would be unsafe to evaluate `conditionC` while at the same time another thread modifies the computation state and affects the truth value of `conditionC`.) The pair of member functions `IloFastMutex::lock` and `IloFastMutex::unlock` delimit the critical section.
- When a thread enters the `wait` call, the mutex is automatically unlocked by the system.
- The loop that repeatedly checks `conditionC` is essential to the correctness of the code fragment. It protects against the following possibility: between the time that a thread modifies the computation state (so that `conditionC` holds) and notifies a waiting thread and the moment the waiting thread wakes up, the computation state might have been changed by another thread, and `conditionC` might very well be false.
- Upon returning from the `wait` call, the mutex is locked. The operation of waking up and locking the mutex is atomic. In other words, nothing can happen between the waking and the locking.

System Class

`IloCondition` is a system class.

Most Concert Technology classes are actually handle classes whose instances point to objects of a corresponding implementation class. For example, instances of the Concert Technology class `IloNumVar` are handles pointing to instances of the implementation class `IloNumVarI`. Their allocation and de-allocation on the Concert Technology heap are managed by an instance of `IloEnv`.

However, system classes, such as `IloCondition`, differ from that Concert Technology pattern. `IloCondition` is an ordinary C++ class. Its instances are allocated on the C++ heap.

Instances of `IloCondition` are not automatically de-allocated by a call to `IloEnv::end`. You must explicitly destroy instances of `IloCondition` by means of a call to the delete operator (which calls the appropriate destructor) when your application no longer needs instances of this class.

Furthermore, you should not allocate—neither directly nor indirectly—any instance of `IloCondition` on the Concert Technology heap because the destructor for that instance of `IloCondition` will never be called automatically by `IloEnv::end` when it cleans up other Concert Technology objects on the Concert Technology heap.

For example, it is not a good idea to make an instance of `IloCondition` part of a conventional Concert Technology model allocated on the Concert Technology heap because that instance will not automatically be de-allocated from the Concert Technology heap along with the other Concert Technology objects.

De-allocating Instances of IloCondition

Instances of `IloCondition` differ from the usual Concert Technology objects because they are not allocated on the Concert Technology heap, and their de-allocation is not managed automatically for you by `IloEnv::end`. Instead, you must explicitly destroy instances of `IloCondition` by calling the delete operator when your application no longer needs those objects.

See Also: IloFastMutex

Constructor and Destructor Summary	
`public`	`IloCondition()`
`public`	`~IloCondition()`

Method Summary	
`public void`	`broadcast()`
`public void`	`notify()`
`public void`	`wait(IloFastMutex * m)`

Constructors and Destructors

`public` **`IloCondition`**`()`

This constructor creates an instance of `IloCondition` and allocates it on the C++ heap (not in a Concert Technology environment). The instance contains data structures specific to an operating system.

`public` **`~IloCondition`**`()`

The delete operator calls this destructor to de-allocate an instance of `IloCondition`. This destructor is called automatically by the runtime system. The destructor de-allocates data structures (specific to an operating system) of the invoking condition.

Methods

`public void` **`broadcast`**`()`

This member function wakes all threads currently waiting on the invoking condition. If there are no threads waiting, this member function does nothing.

`public void` **`notify`**`()`

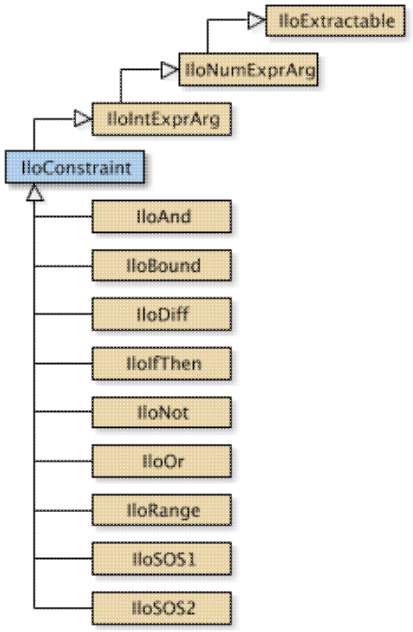
This member function wakes one of the threads currently waiting on the invoking condition.

`public void` **`wait`**`(IloFastMutex * m)`

This member function first puts the calling thread to sleep while it unlocks the mutex `m`. Then, when either of the member functions `broadcast` or `notify` wakes up that thread, this member function acquires the lock on `m` and returns.

Class IloConstraint

Definition file: ilconcert/iloexpression.h



An instance of this class is a constraint in a model.
To create a constraint, you can:

- use a constructor from a subclass of `IloConstraint`, such as `IloRange`, `IloAllDiff`, etc. For example:

```
IloAllDiff allDiff(env, vars);
```

- use a logical operator between constraints to return a constraint. For example, you can use the logical operators on other constraints, like this:

```
IloOr myOr = myConstraint1 || myConstraint2;
```

- use an arithmetic operator between a numeric variable and an expression to return a constraint. For example, you can use the arithmetic operators on numeric variables or expressions, like this:

```
IloRange rng = ( x + 3*y <= 7 );
```

After you create a constraint, you must explicitly add it to the model in order for it to be taken into account. To do so, use the member function `IloModel::add` or the template `IloAdd`. Then extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloConstraintArray, IloModel, IloRange

Constructor Summary	
public	`IloConstraint()`
public	`IloConstraint(IloConstraintI * impl)`

Method Summary	
public IloConstraintI *	getImpl() const

Inherited Methods from IloIntExprArg
getImpl

Inherited Methods from IloNumExprArg
getImpl

Inherited Methods from IloExtractable
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloConstraint**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloConstraint**(IloConstraintI * impl)

This constructor creates a handle object from a pointer to an implementation object.

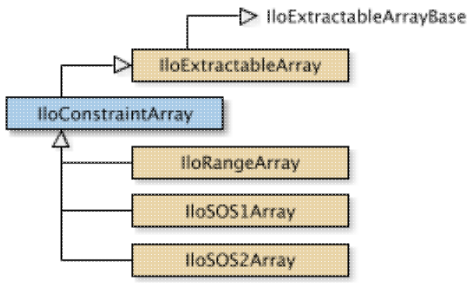
Methods

public IloConstraintI * **getImpl**() const

This member function returns a pointer to the implementation object of the invoking handle.

Class IloConstraintArray

Definition file: ilconcert/iloexpression.h



The array class of constraints for a model.
For each basic type, Concert Technology defines a corresponding array class. `IloConstraintArray` is the array class of constraints for a model.

Instances of `IloConstraintArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Arrays

See Also: IloConstraint, operator>>, operator<<

Constructor Summary	
public	`IloConstraintArray(IloDefaultArrayI * i=0)`
public	`IloConstraintArray(const IloConstraintArray & copy)`
public	`IloConstraintArray(const IloEnv env, IloInt n=0)`

Method Summary	
public void	`add(IloInt more, const IloConstraint x)`
public void	`add(const IloConstraint x)`
public void	`add(const IloConstraintArray x)`
public IloConstraint	`operator[](IloInt i) const`
public IloConstraint &	`operator[](IloInt i)`

Inherited Methods from `IloExtractableArray`
`add, add, add, endElements, setNames`

Constructors

`public` **`IloConstraintArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty array. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

```
public IloConstraintArray(const IloConstraintArray & copy)
```

This copy constructor makes a copy of the array specified by `copy`.

```
public IloConstraintArray(const IloEnv env, IloInt n=0)
```

This constructor creates an array of `n` elements, each of which is an empty handle.

Methods

```
public void add(IloInt more, const IloConstraint x)
```

This member function appends `constraint` to the invoking array multiple times. The argument `more` specifies how many times.

```
public void add(const IloConstraint x)
```

This member function appends `constraint` to the invoking array.

```
public void add(const IloConstraintArray x)
```

This member function appends the elements in `array` to the invoking array.

```
public IloConstraint operator[](IloInt i) const
```

This operator returns a reference to the constraint located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

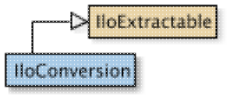
```
 IloConstraint  operator[] (IloInt i) const;
```

```
public IloConstraint & operator[](IloInt i)
```

This operator returns a reference to the constraint located in the invoking array at the position specified by the index `i`.

Class IloConversion

Definition file: ilconcert/iloexpression.h



For IBM ILOG CPLEX: a means to change the type of a numeric variable.
An instance of this class offers you a means to change the type of a numeric variable. For example, in a model (an instance of `IloModel`) extracted for an algorithm (such as an instance of the class `IloCplex`), you may want to convert the type of a given numeric variable (an instance of `IloNumVar`) from `ILOFLOAT` to `ILOINT` or to `ILOBOOL` (or from `IloNumVar::Float` to `IloNumVar::Int` or to `IloNumVar::Bool`). Such a change is known as a conversion.

After you create a conversion, you must explicitly add it to the model in order for it to be taken into account. To do so, use the member function `IloModel::add` or the template `IloAdd`. Then extract the model for an algorithm (such as an instance of `IloCplex`) with the member function `IloAlgorithm::extract`.

Multiple Type Conversions of the Same Variable

You can convert the type of a numeric variable in a model. To do so, create an instance of `IloConversion` and add it to the model. You can also convert the type of a numeric variable after the model has been extracted for an algorithm (such as an instance of `IloCplex`, documented in the IBM ILOG CPLEX Reference Manual).

An instance of `IloCplex` will not accept more than one type conversion of the same variable. That is, you can change the type once, but not twice, in a single instance of `IloCplex`. Attempts to convert the type of the same variable more than once will throw the exception `IloCplex::MultipleConversionException`, documented in the IBM ILOG CPLEX Reference Manual.

In situations where you want to change the type of a numeric variable more than once (for example, from Boolean to integer to floating-point), there are these possibilities:

- You can remove a prior conversion of a given variable in a given model. To do so, use its member function `IloExtractable::end` to delete it and optionally add a new conversion.
- You can apply different conversions to a given variable in more than one model, like this:

```
IloNumVar x(env, 0, 10, ILOBOOL);
IloRange rng = (x <= 10);
IloModel mdl1(env);
mdl1.add(rng);
mdl1.add(IloConversion(env, x, ILOINT));
IloCplex cplex1(mdl1);
IloModel mdl2(env);
mdl2.add(rng);
mdl2.add(IloConversion(env, x, ILOFLOAT));
IloCplex cplex2(mdl2);
```

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also the class `IloCplex` in the *IBM ILOG CPLEX Reference Manual*.

See Also: IloModel

Constructor Summary	
public	IloConversion()
public	IloConversion(IloConversionI * impl)

public	IloConversion(const IloEnv env, const IloNumVar var, IloNumVar::Type t, const char * name=0)
public	IloConversion(const IloEnv env, const IloNumVarArray vars, IloNumVar::Type t, const char * name=0)
public	IloConversion(const IloEnv env, const IloIntVarArray vars, IloNumVar::Type t, const char * name=0)

Method Summary	
public IloConversionI *	getImpl() const

Inherited Methods from IloExtractable
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloConversion**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloConversion**(IloConversionI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IloConversion**(const IloEnv env, const IloNumVar var, IloNumVar::Type t, const char * name=0)

This constructor accepts a numeric variable and a type; it creates a handle to a type conversion to change the type of the variable `var` to the type specified by `t`. You may use the argument `name` to name the type conversion so that you can refer to it by a string identifier.

public **IloConversion**(const IloEnv env, const IloNumVarArray vars, IloNumVar::Type t, const char * name=0)

This constructor accepts an array of numeric variables and a type; it creates a handle to a type conversion to change the type of each variable in the array `vars` to the type specified by `t`. You may use the argument `name` to name the type conversion so that you can refer to it by a string identifier.

public **IloConversion**(const IloEnv env, const IloIntVarArray vars, IloNumVar::Type t, const char * name=0)

This constructor accepts an array of integer variables and a type; it creates a handle to a type conversion to change the type of each variable in the array `vars` to the type specified by `t`. You may use the argument `name` to name the type conversion so that you can refer to it by a string identifier.

Methods

public IloConversionI * **getImpl**() const

This member function returns a pointer to the implementation object of the invoking handle.

Class IloCplex

Definition file: ilcplex/ilocplexi.h

`IloCplex` derives from the class `IloAlgorithm`. Use it to solve Mathematical Programming models, such as:

- LP (linear programming) problems,
- QP (programs with quadratic terms in the objective function),
- QCP (quadratically constrained programming), including the special case of SOCP (second order cone programming) problems, and
- MIP (mixed integer programming) problems.

An algorithm (that is, an instance of `IloAlgorithm`) extracts a model in an environment. The model extracted by an algorithm is known as the active model.

More precisely, models to be solved by `IloCplex` should contain only `IloExtractable` objects from the following list:

- **variables:** objects of type `IloNumVar` and its extensions `IloIntVar` and `IloSemiContVar`
- **range constraints:** objects of type `IloRange`
- **other relational constraints:** objects of type `IloConstraint` of the form *expr1* relation *expr2*, where the relation is one of `==`, `>=`, `<=`, or `!=`
- **objective function:** one object of type `IloObjective`
- **variable type conversions:** objects of type `IloConversion`
- **special ordered sets:** objects of type `IloSOS1` or `IloSOS2`

The expressions used in the constraints and objective function handled by `IloCplex` are built from variables of those listed types and can be linear or quadratic. In addition, expressions may contain the following constructs:

- **minimum:**`IloMin`
- **maximum:**`IloMax`
- **absolute value:**`IloAbs`
- **piecewise linear functions:**`IloPiecewiseLinear`

Expressions that evaluate only to 0 (zero) and 1 (one) are referred to as Boolean expressions. Such expressions also support:

- **negation:**`operator !`
- **conjunction:**`operator &&` or, equivalently, `IloAnd`
- **disjunction:**`operator ||` or, equivalently, `IloOr`

Moreover, Boolean expressions can be constucted not only from variables, but also from constraints.

`IloCplex` will automatically transform all of these constructs into an equivalent representation amenable to `IloCplex`. Such models can be represented in the following way:

```
Minimize (or Maximize)    c'x + x'Qx
subject to                L <= Ax <= U
                 a_i'x + x'Q_i x <= r_i, for i = 1, ..., q
                          l <=  x <= u.
```

That is, in fact, the standard math programming matrix representation that `IloCplex` uses internally. A is the matrix of linear constraint coefficients, and L and U are the vectors of lower and upper bounds on the vector of variables in the array x. The Q matrix must be positive semi-definite (or negative semi-definite in the maximization case) and represents the quadratic terms of the objective function. The matrices Q_i must be positive semi-definite and represent the quadratic terms of the i-th quadratic constraint. The a_i are vectors

containing the correponding linear terms. For details about the Q_i, see the chapter about quadratically constrained programs (QCP) in the *CPLEX User's Manual*.

Special ordered sets (SOS) fall outside the conventional representation in terms of A and Q matrices and are stored separately.

If the model contains integer, Boolean, or semicontinuous variables, or if the model has special ordered sets (SOSs), the model is referred to as a *mixed integer program* (MIP). You can query whether the active model is a MIP with the method `IloCplex::isMIP`.

A model with quadratic terms in the objective is referred to as a *mixed integer quadratic program* (MIQP) if it is also a MIP, and a *quadratic program* (QP) otherwise. You can query whether the active model has a quadratic objective by calling method `IloCplex::isQO`.

A model with quadratic constraints is referred to as a *mixed integer quadratically constrained program* (MIQCP) if it is also a MIP, and as a *quadratically constrained program* (QCP) otherwise. You can query whether the active model is quadratically constrained by calling the method `IloCplex::isQC`. A QCP may or may not have a quadratic objective; that is, a given problem may be both QP and QCP. Likewise, a MIQCP may or may not have a quadratic objective; that is, a given problem may be both MIQP and MIQCP.

If there are no quadratic terms in the objective, no integer constraints, and the problem is not quadratically constrained, and all variables are continuous it is called a *linear program* (LP).

Information related to the matrix representation of the model can be queried through these methods:

- `IloCplex::getNcols` for querying the number of columns of A,
- `IloCplex::getNrows` for querying the number of rows of A; that is, the number of linear constraints,
- `IloCplex::getNQCs` for querying the number of quadratic constraints,
- `IloCplex::getNNZs` for querying the number of nonzero elements in A, and
- `IloCplex::getNSOSs` for querying the number of special ordered sets (SOSs).

Additional information about the active model can be obtained through iterators defined on the different types of modeling objects in the extracted or active model.

`IloCplex` effectively treats all models as MIQCP models. That is, it allows the most general case, although the solution algorithms make efficient use of special cases, such as taking advantage of the absence of quadratic terms in the formulation. The method `IloCplex::solve` begins by solving the *root relaxation* of the MIQCP model, where all integrality constraints and SOSs are ignored. If the model has no integrality constraints or SOSs, then the optimization is complete once the root relaxation is solved. Otherwise, `IloCplex` uses a branch and cut procedure to reintroduce the integrality constraints or SOSs. See the *CPLEX User's Manual* for more information about branch and cut.

Most users can simply call `solve` to solve their models. However, several parameters are available for users who require more control. These parameters are documented in the *CPLEX Parameters Reference Manual*. Perhaps the most important parameter is `IloCplex::RootAlg`, which determines the algorithm used to solve the root relaxation. Possible settings, as defined in the class `IloCplex::Algorithm`, are:

- **IloCplex::Auto** `IloCplex` automatically selects an algorithm. This is the default setting.
- **IloCplex::Primal** Use the primal simplex algorithm. This option is not available for quadratically constrained problems (QCPs).
- **IloCplex::Dual** Use the dual simplex algorithm. This option is not available for quadratically constrained problems (QCPs).
- **IloCplex::Network** Use network simplex on the embedded network part of the model, followed by dual simplex on the entire model. This option is not available for quadratically constrained problems.
- **IloCplex::Barrier** Use the barrier algorithm.
- **IloCplex::Sifting** Use the sifting algorithm. This option is not available for quadratic problems. If selected nonetheless, `IloCplex` defaults to the `IloCplex::Auto` setting.
- **IloCplex::Concurrent** Use the several algorithms concurrently. This option is not available for quadratic problems. If selected nonetheless, `IloCplex` defaults to the `IloCplex::Auto` setting.

Numerous other parameters allow you to control algorithmic aspects of the optimizer. See the nested enumerations `IloCplex::IntParam`, `IloCplex::NumParam`, and `IloCplex#StringParam` for further

information. Parameters are set with the method `setParam`.

Even higher levels of control can be achieved through goals (see `IloCplex::Goal`) or through callbacks (see `IloCplex::Callback` and its extensions).

Information about a Solution

The `solve` method returns an `IloBool` value specifying whether (`IloTrue`) or not (`IloFalse`) a solution (not necessarily the optimal one) has been found. Further information about the solution can be queried with the method `getStatus`. The return code of type `IloAlgorithm::Status` specifies whether the solution is feasible, bounded, or optimal, or if the model has been proven to be infeasible or unbounded.

The method `IloCplex::getCplexStatus` provides more detailed information about the status of the optimizer after `solve` returns. For example, it can provide information on why the optimizer terminated prematurely (time limit, iteration limit, or other similar limits). The methods `IloCplex::isPrimalFeasible` and `IloCplex::isDualFeasible` can determine whether a primal or dual feasible solution has been found and can be queried.

The most important solution information computed by `IloCplex` are usually the solution vector and the objective function value. The method `IloCplex::getValue` queries the solution vector. The method `IloCplex::getObjValue` queries the objective function value. Most optimizers also compute additional solution information, such as dual values, reduced costs, simplex bases, and others. This additional information can also be queried through various methods of `IloCplex`. If you attempt to retrieve solution information that is not available from a particular optimizer, `IloCplex` will throw an exception.

If you are solving an LP and a basis is available, the solution can be further analyzed by performing sensitivity analysis. This information tells you how sensitive the solution is with respect to changes in variable bounds, constraint bounds, or objective coefficients. The information is computed and accessed with the methods `IloCplex::getBoundSA`, `IloCplex::getRangeSA`, `IloCplex::getRHSSA`, and `IloCplex::getObjSA`.

An important consideration when you access solution information is the numeric quality of the solution. Since `IloCplex` performs arithmetic operations using finite precision, solutions are always subject to numeric errors. For most problems, numeric errors are well within reasonable tolerances. However, for numerically difficult models, you are advised to verify the quality of the solution using the method `IloCplex::getQuality`, which offers a variety of quality measures.

More about Solving Problems

By default when the method `IloCplex::solve` is called, `IloCplex` first presolves the model; that is, it transforms the model into a smaller, yet equivalent model. This operation can be controlled with the following parameters:

- `IloCplex::PreInd`,
- `IloCplex::PreDual`,
- `IloCplex::AggInd`, and
- `IloCplex::AggFill`.

For the rare occasion when a user wants to monitor progress during presolve, the callback class `IloCplex::PresolveCallbackI` is provided.

After the presolve is completed, `IloCplex` solves the first node relaxation and (in cases of a true MIP) enters the branch-and-cut process. `IloCplex` provides callback classes that allow the user to monitor solution progress at each level. Callbacks derived from `IloCplex::ContinuousCallbackI` or one of its derived classes are called regularly during the solution of a node relaxation (including the root), and callbacks derived from `IloCplex::MIPCallbackI` or one of its derived callbacks are called regularly during branch-and-cut search. All callbacks provide the option to abort the current optimization.

Branch Priorities and Directions

When a branch occurs at a node in the branch-and-cut tree, usually there is a set of fractional-valued variables available to pick from for branching. `IloCplex` has several built-in rules for making such a choice, and they can be controlled by the parameter `IloCplex::VarSel`. Also, the method `IloCplex::setPriority` allows the

user to specify a priority order. An instance of IloCplex branches on variables with an assigned priority before variables without a priority. It also branches on variables with higher priority before variables with lower priority, when the variables have fractional values.

Frequently, when two new nodes have been created (controlled by the parameter IloCplex::BtTol), one of the two nodes is processed next. This activity is known as diving. The branch direction determines which of the branches, the up or the down branch, is used when diving. By default, IloCplex automatically selects the branch direction. The user can control the branch direction by the method IloCplex::setDirection.

As mentioned before, the greatest flexibility for controlling the branching during branch-and-cut search is provided through goals (see IloCplex::Goal) or through the callbacks (see IloCplex::BranchCallbackI). With these concepts, you can control the branching decision based on runtime information during the search, instead of statically through branch priorities and directions, but the default strategies work well on many problems.

Cuts

An instance of IloCplex can also generate certain cuts in order to strengthen the relaxation, that is, in order to make the relaxation a better approximation of the original MIP. Cuts are constraints added to a model to restrict (cut away) noninteger solutions that would otherwise be solutions of the relaxation. The addition of cuts usually reduces the number of branches needed to solve a MIP.

When solving a MIP, IloCplex tries to generate violated cuts to add to the problem after solving a node. After IloCplex adds cuts, the subproblem is re-optimized. IloCplex then repeats the process of adding cuts at a node and reoptimizing until it finds no further effective cuts.

An instance of IloCplex generates its cuts in such a way that they are valid for all subproblems, even when they are discovered during analysis of a particular node. After a cut has been added to the problem, it will remain in the problem to the end of the optimization. However, cuts are added only internally; that is, they will not be part of the model extracted to the IloCplex object after the optimization. Cuts are most frequently seen at the root node, but they may be added by an instance of IloCplex at other nodes as conditions warrant.

IloCplex looks for various kinds of cuts that can be controlled by the following parameters:

- IloCplex::Cliques,
- IloCplex::Covers,
- IloCplex::FlowCovers,
- IloCplex::GUBCovers,
- IloCplex::FracCuts,
- IloCplex::MIRCuts,
- IloCplex::FlowPaths,
- IloCplex::ImplBd, and
- IloCplex::DisjCuts.

During the search, you can query information about those cuts with a callback (see IloCplex::MIPCallbackI and its subclasses). For types of cuts that may take a long time to generate, callbacks are provided to monitor the progress and potentially abort the cut generation progress. In particular, those callback classes are IloCplex::FractionalCutCallbackI and IloCplex::DisjunctiveCutCallbackI. The callback class IloCplex::CutCallbackI allows you to add your own problem-specific cuts during search. This callback also allows you to generate and add local cuts, that is cuts that are only valid within the subtree where they have been added.

Instead of using callbacks, you can use goals to add your own cuts during the optimization.

Heuristics

After a node has been processed, that is, the LP has been solved and no more cuts were generated, IloCplex may try to construct an integer feasible solution from the LP solution at that node. The parameter IloCplex::HeurFreq and other parameters provide some control over this activity. In addition, goals or the callback class IloCplex::HeuristicCallbackI make it possible to call user-written heuristics to find an integer feasible solution.

Again, instead of using callbacks, you can use goals to add inject your own heuristically constructed solution into the running optimization.

Node Selection

When `IloCplex` is not diving but picking an unexplored node from the tree, several options are available that can be controlled with the parameter `IloCplex::NodeSel`. Again, `IloCplex` offers a callback class, `IloCplex::NodeCallbackI`, to give the user full control over this selection. With goals, objects of type `IloCplex::NodeEvaluatorI` can be used to define your own selection strategy.

See also `IloAlgorithm` in the *CPLEX Reference Manual of the C++ API*.

See also *Goals* among the Concepts in this manual. See also goals in the *CPLEX User's Manual*.

See Also: IloCplex::Algorithm, IloCplex::BasisStatus, IloCplex::BasisStatusArray, IloCplex::BranchDirection, IloCplex::BranchDirectionArray, IloCplex::CallbackI, IloCplex::DeleteMode, IloCplex::DualPricing, IloCplex::Exception, IloCplex::IntParam, IloCplex::MIPEmphasisType, IloCplex::NodeSelect, IloCplex::NumParam, IloCplex::PrimalPricing, IloCplex::Quality, IloCplex::CplexStatus, IloCplex::StringParam, IloCplex::VariableSelect, IloCplex::GoalI

Attribute Summary	
`public static const int`	`IncumbentId`

Constructor Summary	
`public`	`IloCplex(IloEnv env)`
`public`	`IloCplex(const IloModel model)`

Method Summary	
`public IloConstraint`	`addCut(IloConstraint con)`
`public const IloConstraintArray`	`addCuts(const IloConstraintArray con)`
`public IloCplex::FilterIndex`	`addDiversityFilter(IloNum lower_cutoff, IloNum upper_cutoff, const IloIntVarArray vars, const IloNumArray weights, const IloNumArray refval, const char * fname=0)`
`public IloCplex::FilterIndex`	`addDiversityFilter(IloNum lower_cutoff, IloNum upper_cutoff, const IloNumVarArray vars, const IloNumArray weights, const IloNumArray refval, const char * fname=0)`
`public IloConstraint`	`addLazyConstraint(IloConstraint con)`
`public const IloConstraintArray`	`addLazyConstraints(const IloConstraintArray con)`
`public IloInt`	`addMIPStart(IloNumVarArray vars=0, IloNumArray values=0, IloCplex::MIPStartEffort effort=MIPStartAuto, const char * name=0)`
`public IloCplex::FilterIndex`	`addRangeFilter(IloNum, IloNum, const IloIntVarArray, const IloNumArray, const char *=0)`
`public IloCplex::FilterIndex`	`addRangeFilter(IloNum, IloNum, const IloNumVarArray, const IloNumArray, const char *=0)`
`public IloConstraint`	`addUserCut(IloConstraint con)`
`public const IloConstraintArray`	`addUserCuts(const IloConstraintArray con)`

public static IloCplex::Goal	Apply(IloCplex cplex, IloCplex::Goal goal, IloCplex::NodeEvaluator eval)
public void	basicPresolve(const IloIntVarArray vars, IloNumArray redlb=0, IloNumArray redub=0, const IloRangeArray rngs=0, IloBoolArray redundant=0) const
public void	basicPresolve(const IloNumVarArray vars, IloNumArray redlb=0, IloNumArray redub=0, const IloRangeArray rngs=0, IloBoolArray redundant=0) const
public void	changeMIPStart(IloInt mipstartindex, IloNumVarArray vars, IloNumArray values, IloCplex::MIPStartEffort effortlevel=MIPStartAuto)
public void	clearCuts()
public void	clearLazyConstraints()
public void	clearModel()
public void	clearUserCuts()
public void	delDirection(IloIntVar var)
public void	delDirection(IloNumVar var)
public void	delDirections(const IloIntVarArray var)
public void	delDirections(const IloNumVarArray var)
public void	deleteMIPStarts(IloInt first, IloInt num=1)
public void	delFilter(IloCplex::FilterIndex filter)
public void	delPriorities(const IloIntVarArray var)
public void	delPriorities(const IloNumVarArray var)
public void	delPriority(IloIntVar var)
public void	delPriority(IloNumVar var)
public void	delSolnPoolSoln(IloInt which)
public void	delSolnPoolSolns(IloInt begin, IloInt end)
public IloNum	dualFarkas(IloConstraintArray rng, IloNumArray y)
public void	exportModel(const char * filename) const
public IloBool	feasOpt(const IloConstraintArray cts, const IloNumArray prefs)
public IloBool	feasOpt(const IloRangeArray rngs, const IloNumArray rnglb, const IloNumArray rngub)
public IloBool	feasOpt(const IloIntVarArray vars, const IloNumArray varlb, const IloNumArray varub)
public IloBool	feasOpt(const IloNumVarArray vars, const IloNumArray varlb, const IloNumArray varub)
public IloBool	feasOpt(const IloRangeArray rngs, const IloNumArray rnglb, const IloNumArray rngub, const IloIntVarArray vars, const IloNumArray varlb, const IloNumArray varub)
public IloBool	feasOpt(const IloRangeArray rngs, const IloNumArray rnglb, const IloNumArray rngub, const IloNumVarArray vars, const IloNumArray

	varlb, const IloNumArray varub)
public void	freePresolve()
public IloCplex::Aborter	getAborter()
public IloCplex::Algorithm	getAlgorithm() const
public void	getAX(IloNumArray val, const IloRangeArray con) const
public IloNum	getAX(const IloRange range) const
public IloCplex::BasisStatus	getBasisStatus(const IloConstraint con) const
public IloCplex::BasisStatus	getBasisStatus(const IloIntVar var) const
public IloCplex::BasisStatus	getBasisStatus(const IloNumVar var) const
public void	getBasisStatuses(IloCplex::BasisStatusArray cstat, const IloIntVarArray var, IloCplex::BasisStatusArray rstat, const IloConstraintArray con) const
public void	getBasisStatuses(IloCplex::BasisStatusArray cstat, const IloNumVarArray var, IloCplex::BasisStatusArray rstat, const IloConstraintArray con) const
public void	getBasisStatuses(IloCplex::BasisStatusArray stat, const IloConstraintArray con) const
public void	getBasisStatuses(IloCplex::BasisStatusArray stat, const IloIntVarArray var) const
public void	getBasisStatuses(IloCplex::BasisStatusArray stat, const IloNumVarArray var) const
public IloNum	getBestObjValue() const
public void	getBoundSA(IloNumArray lblower, IloNumArray lbupper, IloNumArray ublower, IloNumArray ubupper, const IloIntVarArray vars) const
public void	getBoundSA(IloNumArray lblower, IloNumArray lbupper, IloNumArray ublower, IloNumArray ubupper, const IloNumVarArray vars) const
public IloCplex::ConflictStatus	getConflict(IloConstraint con) const
public IloCplex::ConflictStatusArray	getConflict(IloConstraintArray cons) const
public IloCplex::CplexStatus	getCplexStatus() const
public IloCplex::CplexStatus	getCplexSubStatus() const
public IloNum	getCplexTime() const
public IloNum	getCutoff() const
public const char *	getDefault(IloCplex::StringParam parameter) const
public IloNum	getDefault(IloCplex::NumParam parameter) const
public IloInt	getDefault(IloCplex::IntParam parameter) const
public IloBool	getDefault(IloCplex::BoolParam parameter) const
public IloCplex::DeleteMode	getDeleteMode() const
public IloCplex::BranchDirection	getDirection(IloIntVar var) const

public IloCplex::BranchDirection	getDirection(IloNumVar var) const
public void	getDirections(IloCplex::BranchDirectionArray dir, const IloIntVarArray var) const
public void	getDirections(IloCplex::BranchDirectionArray dir, const IloNumVarArray var) const
public IloExtractable	getDiverging() const
public IloNum	getDiversityFilterLowerCutoff(IloCplex::FilterIndex filter) const
public void	getDiversityFilterRefVals(IloCplex::FilterIndex filter, IloNumArray) const
public IloNum	getDiversityFilterUpperCutoff(IloCplex::FilterIndex filter) const
public void	getDiversityFilterWeights(IloCplex::FilterIndex filter, IloNumArray) const
public IloNum	getDual(const IloRange range) const
public void	getDuals(IloNumArray val, const IloRangeArray con) const
public IloCplex::FilterIndex	getFilterIndex(const char * lname_str) const
public IloCplex::FilterType	getFilterType(IloCplex::FilterIndex filter) const
public void	getFilterVars(IloCplex::FilterIndex filter, IloNumVarArray) const
public IloInt	getIncumbentNode() const
public IloNum	getInfeasibilities(IloNumArray infeas, const IloIntVarArray var) const
public IloNum	getInfeasibilities(IloNumArray infeas, const IloNumVarArray var) const
public IloNum	getInfeasibilities(IloNumArray infeas, const IloConstraintArray con) const
public IloNum	getInfeasibility(const IloIntVar var) const
public IloNum	getInfeasibility(const IloNumVar var) const
public IloNum	getInfeasibility(const IloConstraint con) const
public IloNum	getMax(IloCplex::NumParam parameter) const
public IloInt	getMax(IloCplex::IntParam parameter) const
public IloNum	getMin(IloCplex::NumParam parameter) const
public IloInt	getMin(IloCplex::IntParam parameter) const
public IloNum	getMIPRelativeGap() const
public IloCplex::MIPStartEffort	getMIPStart(IloInt mipstartindex, const IloNumVarArray vars, IloNumArray vals, IloBoolArray isset)
public IloInt	getMIPStartIndex(const char * lname_str) const
public const char *	getMIPStartName(IloInt mipstartindex)
public IloInt	getNbarrierIterations() const
public IloInt	getNbinVars() const
public IloInt	getNcols() const

public IloInt	getNcrossDExch() const
public IloInt	getNcrossDPush() const
public IloInt	getNcrossPExch() const
public IloInt	getNcrossPPush() const
public IloInt	getNcuts(IloCplex::CutType which) const
public IloInt	getNdualSuperbasics() const
public IloInt	getNfilters() const
public IloInt	getNintVars() const
public IloInt	getNiterations() const
public int	getNMIPStarts() const
public IloInt	getNnodes() const
public IloInt	getNnodesLeft() const
public IloInt	getNNZs() const
public IloInt	getNphaseOneIterations() const
public IloInt	getNprimalSuperbasics() const
public IloInt	getNQCs() const
public IloInt	getNrows() const
public IloInt	getNsemiContVars() const
public IloInt	getNsemiIntVars() const
public IloInt	getNsiftingIterations() const
public IloInt	getNsiftingPhaseOneIterations() const
public IloInt	getNSOSs() const
public IloObjective	getObjective() const
public void	getObjSA(IloNumArray lower, IloNumArray upper, const IloIntVarArray cols) const
public void	getObjSA(IloNumArray lower, IloNumArray upper, const IloNumVarArray vars) const
public IloNum	getObjValue(IloInt soln) const
public const char *	getParam(IloCplex::StringParam parameter) const
public IloNum	getParam(IloCplex::NumParam parameter) const
public IloBool	getParam(IloCplex::BoolParam parameter) const
public IloInt	getParam(IloCplex::IntParam parameter) const
public IloCplex::ParameterSet	getParameterSet()
public void	getPriorities(IloNumArray pri, const IloIntVarArray var) const
public void	getPriorities(IloNumArray pri, const IloNumVarArray var) const
public IloNum	getPriority(IloIntVar var) const
public IloNum	getPriority(IloNumVar var) const
public IloNum	getQuality(IloCplex::Quality q, IloInt soln, IloNumVar * var=0, IloConstraint * rng=0) const

public IloNum	getQuality(IloCplex::Quality q, IloInt soln, IloConstraint * rng, IloNumVar * var=0) const
public IloNum	getQuality(IloCplex::Quality q, IloConstraint * rng, IloNumVar * var=0) const
public IloNum	getQuality(IloCplex::Quality q, IloNumVar * var=0, IloConstraint * rng=0) const
public void	getRangeFilterCoefs(IloCplex::FilterIndex filter, IloNumArray) const
public IloNum	getRangeFilterLowerBound(IloCplex::FilterIndex filter) const
public IloNum	getRangeFilterUpperBound(IloCplex::FilterIndex filter) const
public void	getRangeSA(IloNumArray lblower, IloNumArray lbupper, IloNumArray ublower, IloNumArray ubupper, const IloRangeArray con) const
public void	getRay(IloNumArray vals, IloNumVarArray vars) const
public IloNum	getReducedCost(const IloIntVar var) const
public IloNum	getReducedCost(const IloNumVar var) const
public void	getReducedCosts(IloNumArray val, const IloIntVarArray var) const
public void	getReducedCosts(IloNumArray val, const IloNumVarArray var) const
public void	getRHSSA(IloNumArray lower, IloNumArray upper, const IloRangeArray cons) const
public IloNum	getSlack(const IloRange range, IloInt soln=IncumbentId) const
public void	getSlacks(IloNumArray val, const IloRangeArray con, IloInt soln=IncumbentId) const
public IloNum	getSolnPoolMeanObjValue() const
public IloInt	getSolnPoolNreplaced() const
public IloInt	getSolnPoolNsolns() const
public IloAlgorithm::Status	getStatus() const
public IloCplex::Algorithm	getSubAlgorithm() const
public IloNum	getValue(const IloObjective ob, IloInt soln) const
public IloNum	getValue(const IloNumExprArg expr, IloInt soln) const
public IloNum	getValue(const IloIntVar var, IloInt soln) const
public IloNum	getValue(const IloNumVar var, IloInt soln) const
public void	getValues(const IloIntVarArray var, IloNumArray val, IloInt soln) const
public void	getValues(IloNumArray val, const IloIntVarArray var, IloInt soln) const

public void	getValues(IloNumArray val, const IloNumVarArray var, IloInt soln) const
public void	getValues(const IloIntVarArray var, IloNumArray val) const
public void	getValues(IloNumArray val, const IloIntVarArray var) const
public void	getValues(const IloNumVarArray var, IloNumArray val) const
public void	getValues(IloNumArray val, const IloNumVarArray var) const
public const char *	getVersion() const
public void	importModel(IloModel & m, const char * filename) const
public void	importModel(IloModel & m, const char * filename, IloObjective & obj, IloNumVarArray vars, IloRangeArray rngs, IloRangeArray lazy=0, IloRangeArray cuts=0) const
public void	importModel(IloModel & model, const char * filename, IloObjective & obj, IloNumVarArray vars, IloRangeArray rngs, IloSOS1Array sos1, IloSOS2Array sos2, IloRangeArray lazy=0, IloRangeArray cuts=0) const
public IloBool	isDualFeasible() const
public IloBool	isMIP() const
public IloBool	isPrimalFeasible() const
public IloBool	isQC() const
public IloBool	isQO() const
public static IloCplex::Goal	LimitSearch(IloCplex cplex, IloCplex::Goal goal, IloCplex::SearchLimit limit)
public IloBool	populate()
public void	presolve(IloCplex::Algorithm alg)
public void	protectVariables(const IloIntVarArray var)
public void	protectVariables(const IloNumVarArray var)
public void	qpIndefCertificate(IloIntVarArray var, IloNumArray x)
public void	qpIndefCertificate(IloNumVarArray var, IloNumArray x)
public void	readBasis(const char * name) const
public IloCplex::FilterIndexArray	readFilters(const char * name)
public void	readMIPStart(const char * name) const
public void	readMIPStarts(const char * name) const
public void	readOrder(const char * filename) const
public void	readParam(const char * name) const
public void	readSolution(const char * name) const
public IloBool	refineConflict(IloConstraintArray cons, IloNumArray prefs)
public IloBool	refineMIPStartConflict(IloInt mipstartindex,

	IloConstraintArray cons, IloNumArray prefs)
public void	remove(IloCplex::Aborter abort)
public void	setBasisStatuses(const IloCplex::BasisStatusArray cstat, const IloIntVarArray var, const IloCplex::BasisStatusArray rstat, const IloConstraintArray con)
public void	setBasisStatuses(const IloCplex::BasisStatusArray cstat, const IloNumVarArray var, const IloCplex::BasisStatusArray rstat, const IloConstraintArray con)
public void	setDefaults()
public void	setDeleteMode(IloCplex::DeleteMode mode)
public void	setDirection(IloIntVar var, IloCplex::BranchDirection dir)
public void	setDirection(IloNumVar var, IloCplex::BranchDirection dir)
public void	setDirections(const IloIntVarArray var, const IloCplex::BranchDirectionArray dir)
public void	setDirections(const IloNumVarArray var, const IloCplex::BranchDirectionArray dir)
public void	setParam(IloCplex::StringParam parameter, const char * value)
public void	setParam(IloCplex::NumParam parameter, IloNum value)
public void	setParam(IloCplex::BoolParam parameter, IloBool value)
public void	setParam(IloCplex::IntParam parameter, IloInt value)
public void	setParameterSet(IloCplex::ParameterSet set)
public void	setPriorities(const IloIntVarArray var, const IloNumArray pri)
public void	setPriorities(const IloNumVarArray var, const IloNumArray pri)
public void	setPriority(IloIntVar var, IloNum pri)
public void	setPriority(IloNumVar var, IloNum pri)
public void	setVectors(const IloNumArray x, const IloNumArray dj, const IloIntVarArray var, const IloNumArray slack, const IloNumArray pi, const IloRangeArray rng)
public void	setVectors(const IloNumArray x, const IloNumArray dj, const IloNumVarArray var, const IloNumArray slack, const IloNumArray pi, const IloRangeArray rng)
public IloBool	solve(IloCplex::Goal goal)
public IloBool	solve()
public IloBool	solveFixed(IloInt soln=IloCplex::IncumbentId)
public IloInt	tuneParam(IloArray< const char * > filename)

public IloInt	tuneParam(IloCplex::ParameterSet fixedset)
public IloInt	tuneParam()
public IloInt	tuneParam(IloArray< const char * > filename, IloCplex::ParameterSet fixedset)
public IloCplex::Callback	use(IloCplex::Callback cb)
public IloCplex::Aborter	use(IloCplex::Aborter abort)
public void	writeBasis(const char * name) const
public void	writeConflict(const char * filename) const
public void	writeFilters(const char * name)
public void	writeMIPStart(const char * name, IloInt mst=0) const
public void	writeMIPStarts(const char * name, IloInt first=0, IloInt num=IloIntMax) const
public void	writeOrder(const char * filename) const
public void	writeParam(const char * name) const
public void	writeSolution(const char * name, IloInt soln=IncumbentId) const
public void	writeSolutions(const char * name) const

Inherited Methods from `IloAlgorithm`
clear, end, error, extract, getEnv, getIntValue, getIntValues, getModel, getObjValue, getStatus, getTime, getValue, getValue, getValue, getValue, getValues, getValues, isExtracted, out, printTime, resetTime, setError, setOut, setWarning, solve, warning

Inner Enumeration
IloCplex::Algorithm
IloCplex::BasisStatus
IloCplex::BoolParam
IloCplex::BranchDirection
IloCplex::ConflictStatus
IloCplex::CplexStatus
IloCplex::CutType
IloCplex::DeleteMode
IloCplex::DualPricing
IloCplex::IntParam
IloCplex::MIPEmphasisType
IloCplex::MIPsearch
IloCplex::MIPStartEffort
IloCplex::NodeSelect
IloCplex::NumParam
IloCplex::Parallel_Mode
IloCplex::PrimalPricing
IloCplex::Quality

IloCplex::Relaxation
IloCplex::StringParam
IloCplex::TuningStatus
IloCplex::VariableSelect
IloCplex::WriteLevelType

Inner Typedef	
IloCplex::BasisStatusArray	
IloCplex::BranchDirectionArray	
IloCplex::ConflictStatusArray	
IloCplex::Status	An enumeration for the class `IloAlgorithm`.

Inner Class
IloCplex::Aborter
IloCplex::BarrierCallbackI
IloCplex::BranchCallbackI
IloCplex::Callback
IloCplex::CallbackI
IloCplex::ContinuousCallbackI
IloCplex::ControlCallbackI
IloCplex::CrossoverCallbackI
IloCplex::CutCallbackI
IloCplex::DisjunctiveCutCallbackI
IloCplex::DisjunctiveCutInfoCallbackI
IloCplex::Exception
IloCplex::FlowMIRCutCallbackI
IloCplex::FlowMIRCutInfoCallbackI
IloCplex::FractionalCutCallbackI
IloCplex::FractionalCutInfoCallbackI
IloCplex::Goal
IloCplex::GoalI
IloCplex::HeuristicCallbackI
IloCplex::IncumbentCallbackI
IloCplex::InvalidCutException
IloCplex::LazyConstraintCallbackI
IloCplex::MIPCallbackI
IloCplex::MIPInfoCallbackI
IloCplex::MultipleConversionException
IloCplex::MultipleObjException
IloCplex::NetworkCallbackI
IloCplex::NodeCallbackI
IloCplex::NodeEvaluator

IloCplex::NodeEvaluatorI
IloCplex::OptimizationCallbackI
IloCplex::ParameterSet
IloCplex::PresolveCallbackI
IloCplex::ProbingCallbackI
IloCplex::ProbingInfoCallbackI
IloCplex::SearchLimit
IloCplex::SearchLimitI
IloCplex::SimplexCallbackI
IloCplex::SolveCallbackI
IloCplex::TuningCallbackI
IloCplex::UnknownExtractableException
IloCplex::UserCutCallbackI

Attributes

`public static const int` **`IncumbentId`**

This constant identifies the incumbent solution for a MIP in methods which require a solution index.

Constructors

`public` **`IloCplex`**`(IloEnv env)`

This constructor creates a CPLEX algorithm. The new `IloCplex` object has no `IloModel` loaded (or extracted) to it.

`public` **`IloCplex`**`(const IloModel model)`

This constructor creates a CPLEX algorithm and extracts `model` for that algorithm.

When you create an algorithm (an instance of `IloCplex`, for example) and extract a model for it, you can write either this line:

```
IloCplex cplex(model);
```

or these two lines:

```
IloCplex cplex(env);
cplex.extract(model);
```

Methods

`public IloConstraint` **`addCut`**`(IloConstraint con)`

This method adds `con` as a cut to the invoking `IloCplex` object. The cut is not extracted as the regular

constraints in a model, but is only copied when invoking the method `addCut`. Thus, `con` may be deleted or modified after `addCut` has been called and the change will not be notified to the invoking `IloCplex` object.

When columns are deleted from the extracted model, all cuts are deleted as well and need to be reextracted if they should be considered. Cuts are not part of the root problem, but are considered on an as-needed basis. A solution computed by `IloCplex` is guaranteed to satisfy all cuts added with this method.

```
public const IloConstraintArray addCuts(const IloConstraintArray con)
```

This method adds the constraints in `con` as cuts to the invoking `IloCplex` object. Everything said for `IloCplex::addCut` applies equally to each of the cuts given in array `con`.

```
public IloCplex::FilterIndex addDiversityFilter(IloNum lower_cutoff, IloNum
upper_cutoff, const IloIntVarArray vars, const IloNumArray weights, const
IloNumArray refval, const char * fname=0)
```

Creates and installs a named diversity filter for the designated integer variables with the specified lower and upper cutoff values, reference values, and weights.

A *diversity filter* drives the search for multiple solutions toward new solutions that satisfy a measure of diversity specified in the filter.

This diversity measure applies only to binary variables.

Potential new solutions are compared to a reference set. You must specify which variables are to be compared. You do so with the argument `vars` designating the indices of variables to include in the diversity measure.

A *reference set* is the set of values specified by the argument `refval`.

You may optionally specify weights (that is, coefficients to form a linear expression in terms of the variables) in the diversity measure; if you do not specify weights, all differences between the reference set and potential new solutions will be weighted by the value 1.0 (one). CPLEX computes the diversity measure by summing the pair-wise weighted absolute differences from the reference values, like this:

```
differences(x) = sum {weight[i] times |x[vars[i]] - refval[i]|}.
```

A diversity filter makes sure that the solutions satisfy the constraint:

```
lower bound <= differences(x) <= upper bound
```

You may specify both a lower and upper bound on diversity.

In order to say, *Give me solutions that are close to this one, within this specified set of variables,* specify a `lower_bound` of 0.0 (zero) and a finite `upper_bound`. CPLEX then looks for solutions that differ from the reference values by at most the value of `upper_bound`, within the specified set of variables.

In order to say, *Give me solutions that are different from this one,* specify a finite `lower_bound` and an infinite (that is, very large) `upper_bound` on the diversity. CPLEX then looks for solutions that differ from the reference values by at least the value of `lower_bound`, within the specified set of variables.

Parameters:

vars	An array of integer variables in the diversity measure.
weights	An array of weights corresponding to the variables to be used in the diversity measure. May be NULL, in which case CPLEX uses weights of 1.0 (one).
refval	

	An array of reference values for the the variables in the diversity filter to compare with a solution when CPLEX computes the diversity measure.
fname	The name of the filter. May be NULL.

Returns:

This method returns the index of the added filter.

```
public IloCplex::FilterIndex addDiversityFilter(IloNum lower_cutoff, IloNum
upper_cutoff, const IloNumVarArray vars, const IloNumArray weights, const
IloNumArray refval, const char * fname=0)
```

This method creates and installs a named diversity filter for the designated numeric variables with the specified lower and upper bounds, reference values, and weights.

A *diversity filter* drives the search for multiple solutions toward new solutions that satisfy a measure of diversity specified in the filter.

This diversity measure applies only to binary variables.

Potential new solutions are compared to a reference set. You must specify which variables are to be compared. You do so with the argument `vars` designating the indices of variables to include in the diversity measure.

A *reference set* is the set of values specified by the argument `refval`.

You may optionally specify weights (that is, coefficients to form a linear expression in terms of the variables) in the diversity measure; if you do not specify weights, all differences between the reference set and potential new solutions will be weighted by the value 1.0 (one). CPLEX computes the diversity measure by summing the pair-wise weighted absolute differences from the reference values, like this:

```
differences(x) = sum {weight[i] times |x[vars[i]] - refval[i]|}.
```

A diversity filter makes sure that the solutions satisfy the constraint:

```
lower bound <= differences(x) <= upper bound
```

You may specify both a lower and upper bound on diversity.

In order to say, *Give me solutions that are close to this one, within this specified set of variables,* specify a `lower_bound` of 0.0 (zero) and a finite `upper_bound`. CPLEX then looks for solutions that differ from the reference values by at most the value of `upper_bound`, within the specified set of variables.

In order to say, *Give me solutions that are different from this one,* specify a finite `lower_bound` and an infinite (that is, very large) `upper_bound` on the diversity. CPLEX then looks for solutions that differ from the reference values by at least the value of `lower_bound`, within the specified set of variables.

Parameters:

vars	An array of numeric variables in the diversity measure.
weights	An array of weights corresponding to the variables to be used in the diversity measure. May be NULL, in which case CPLEX uses weights of 1.0 (one).
refval	An array of reference values for the the variables in the diversity filter to compare with a solution when CPLEX computes the diversity measure.
fname	The name of the filter. May be NULL.

Returns:

This method returns the index of the added filter.

```
public IloConstraint addLazyConstraint(IloConstraint con)
```

Note

This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method adds `con` as a lazy constraint to the invoking `IloCplex` object. The constraint `con` is copied into the lazy constraint pool; the `con` itself is not part of the pool, so changes to `con` after it has been copied into the lazy constraint pool will not affect the lazy constraint pool.

Lazy constraints added with `addLazyConstraint` are typically constraints of the model that are not expected to be violated when left out. The idea behind this is that the LPs that are solved when solving the MIP can be kept smaller when these constraints are not included. `IloCplex` will, however, include a lazy constraint in the LP as soon as it becomes violated. In other words, the solution computed by `IloCplex` makes sure that all the lazy constraints that have been added are satisfied.

By contrast, if the constraint does not change the feasible region of the extracted model but only strengthens the formulation, it is referred to as a user cut. While user cuts can be added to `IloCplex` with `addLazyConstraint`, it is generally preferable to do so with `addUserCuts`. It is an error, however, to add lazy constraints by means of the method `addUserCuts`.

When columns are deleted from the extracted model, all lazy constraints are deleted as well and need to be recopied into the lazy constraint pool. Use of this method in place of `addCuts` allows for further presolve reductions

This method is equivalent to `IloCplex::addCut`.

```
public const IloConstraintArray addLazyConstraints(const IloConstraintArray con)
```

Note

This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method adds a set of lazy constraints to the invoking `IloCplex` object. Everything said for `IloCplex::addLazyConstraint` applies to each of the lazy constraints given in array `con`.

This method is equivalent to `IloCplex::addCuts`.

```
public IloInt addMIPStart(IloNumVarArray vars=0, IloNumArray values=0,
IloCplex::MIPStartEffort effort=MIPStartAuto, const char * name=0)
```

Adds a named MIP start with the specified level of effort defined by the specified variables and their corresponding values to the current problem.

```
public IloCplex::FilterIndex addRangeFilter(IloNum, IloNum, const IloIntVarArray,
const IloNumArray, const char *=0)
```

Creates a named range filter, using the specified lower bound, upper bound, integer variables, and weights, adds the filter to the solution pool of the invoking model, and returns the index of the filter.

A range filter drives the search for multiple solutions toward new solutions that satisfy criteria specified as a ranged linear expression in the filter. A range filter sets a lower and an upper bound on a linear expression consisting of variables designated in the array `vars` and coefficient values designated in the argument `weights`, like this:

```
lower bound <= sum{weights[i] times vars[i]}  <= upper bound
```

A range filter applies to variables of any type; that is, binary, general integer, continuous.

Returns:

This method returns the index of the added filter.

```
public IloCplex::FilterIndex addRangeFilter(IloNum, IloNum, const IloNumVarArray,
const IloNumArray, const char *=0)
```

This method creates a named range filter, using the specified lower cutoff, upper cutoff, numeric variables, and weights, adds the filter to the solution pool of the invoking model, and returns its index.

A range filter drives the search for multiple solutions toward new solutions that satisfy criteria specified as a ranged linear expression in the filter. A range filter sets a lower and an upper bound on a linear expression consisting of variables designated in the array `vars` and coefficient values designated in the argument `weights`, like this:

```
lower bound <= sum{weights[i] times vars[i]}  <= upper bound
```

A range filter applies to variables of any type; that is, binary, general integer, continuous.

Returns:

This method returns the index of the added filter.

```
public IloConstraint addUserCut(IloConstraint con)
```

Note

This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method adds `con` as a user cut to the invoking `IloCplex` object. The constraint `con` is copied into the user cut pool; the `con` itself is not part of the pool, so changes to `con` after it has been copied into the user cut pool will not affect the user cut pool.

Cuts added with `addUserCut` must be real cuts, in that the solution of a MIP does not depend on whether the cuts are added or not. Instead, they are there only to strengthen the formulation.

When columns are deleted from the extracted model, all user cuts are deleted as well and need to be recopied into the user cut pool.

Note

It is an error to use `addUserCut` for lazy constraints, that is, constraints whose absence may potentially change the solution of the problem. Use `addLazyConstraints` or, equivalently, `addCut` when you add such

a constraint.

```
public const IloConstraintArray addUserCuts(const IloConstraintArray con)
```

Note

This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method adds a set of user cuts to the invoking `IloCplex` object. Everything said for `IloCplex::addUserCut` applies to each of the user cuts given in array `con`.

```
public static IloCplex::Goal Apply(IloCplex cplex, IloCplex::Goal goal,
IloCplex::NodeEvaluator eval)
```

This method is used to create and return a goal that applies the node selection strategy defined by `eval` to the search strategy defined by `goal`. The resulting goal will use the node strategy defined by `eval` for the subtree generated by `goal`.

```
public void basicPresolve(const IloNumVarArray vars, IloNumArray redlb=0,
IloNumArray redub=0, const IloRangeArray rngs=0, IloBoolArray redundant=0) const
public void basicPresolve(const IloIntVarArray vars, IloNumArray redlb=0,
IloNumArray redub=0, const IloRangeArray rngs=0, IloBoolArray redundant=0) const
```

This method can be used to compute tighter bounds for the variables of a model and to detect redundant constraints in the model extracted to the invoking `IloCplex` object. For every variable specified by the argument `vars`, this method will return possibly tightened bounds in the corresponding elements of arrays `redlb` and `redub`. Similarly, for every constraint specified by the argument `rngs`, this method will return a Boolean value reporting whether or not the constraint is redundant in the model in the corresponding element of array `redundant`.

For a semicontinuous or semi-integer variable, this method produces the lower bound of the variable, **not** the semicontinuous or semi-integer lower bound. If this method produces a lower bound less than or equal to zero, then the variable persists as a semicontinuous or semi-integer variable. In contrast, if this method produces a lower bound strictly greater than zero, then basic presolve has concluded that zero can be eliminated from the domain of the variable. Consequently, it is possible to change the type of the variable from semicontinuous to continuous or from semi-integer to integer. Afterwards, you can use the tightened bound without affecting the feasible region of the model.

```
public void changeMIPStart(IloInt mipstartindex, IloNumVarArray vars, IloNumArray
values, IloCplex::MIPStartEffort effortlevel=MIPStartAuto)
```

Changes the MIP start designated by its index by assigning corresponding values to the designated variables and by associating the specified level of effort.

```
public void clearCuts()
```

This method deletes all cuts that have previously been added to the invoking `IloCplex` object with the methods `addCut` and `addCuts`.

```
public void clearLazyConstraints()
```

Note

This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method deletes all lazy constraints added to the invoking `IloCplex` object with the methods `IloCplex::addLazyConstraint` and `IloCplex::addLazyConstraints`.

This method is equivalent to `IloCplex::clearCuts`.

```
public void clearModel()
```

This method can be used to unextract the model that is currently extracted to the invoking `IloCplex` object.

```
public void clearUserCuts()
```

Note

This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method deletes all user cuts that have previously been added to the invoking `IloCplex` object with the methods `IloCplex::addUserCut` and `IloCplex::addUserCuts`.

```
public void delDirection(IloNumVar var)
public void delDirection(IloIntVar var)
```

This method removes any existing branching direction assignment from variable `var`.

```
public void delDirections(const IloNumVarArray var)
public void delDirections(const IloIntVarArray var)
```

This method removes any existing branching direction assignments from all variables in the array `var`.

```
public void deleteMIPStarts(IloInt first, IloInt num=1)
```

Deletes the designated number of MIP starts, starting from the MIP start specified by its index.

```
public void delFilter(IloCplex::FilterIndex filter)
```

Deletes the speficied filter from the solution pool.

```
public void delPriorities(const IloNumVarArray var)
public void delPriorities(const IloIntVarArray var)
```

This method removes any existing priority order assignments from all variables in the array `var`.

```
public void delPriority(IloNumVar var)
public void delPriority(IloIntVar var)
```

This method removes any existing priority order assignment from variable `var`.

```
public void delSolnPoolSoln(IloInt which)
```

Deletes the specified solution from the solution pool and renumbers the indices of the remaining solutions in the pool.

```
public void delSolnPoolSolns(IloInt begin, IloInt end)
```

Deletes a range of solutions from the solution pool and renumbers the indices of the remaining solutions in the pool.

```
public IloNum dualFarkas(IloConstraintArray rng, IloNumArray y)
```

Note

This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method returns a Farkas proof of infeasibility for the active LP model after it has been proven to be infeasible by one of the simplex optimizers. For every constraint `i` of the active LP this method computes a value `y[i]` such that `y'A >= y'b`, where `A` denotes the constraint matrix. For more detailed information about the Farkas proof of infeasibility, see the C routine `CPXdualfarkas`, documented in the reference manual of the Callable Library.

Parameters:

rng An array of length `getNrows` where constraints corresponding to the values in `y` are returned.

y An array of length `getNrows`.

Returns:

The value of `y'b - y'A z` for the vector `z` defined such that `z[j] = ub[j]` if `y'A[j] > 0` and `z[j] = lb[j]` if `y'A[j] < 0` for all variables `j`.

```
public void exportModel(const char * filename) const
```

This method writes the active model (that is, the model that has been extracted by the invoking algorithm) to the file `filename`. The file format is determined by the extension of the file name. The following extensions are recognized on most platforms:

- .sav
- .mps
- .lp
- .sav.gz (if gzip is properly installed)
- .mps.gz (if gzip is properly installed)
- .lp.gz (if gzip is properly installed)

Microsoft Windows does not support gzipped files for this API.

If no name has been assigned to a variable or range (that is, the method IloExtractable::getName returns null for that variable or range), IloCplex uses a default name when writing the model to the file (or to the optimization log). Default names are of the form IloXj for variables and IloCi, where i and j are internal indices of IloCplex.

See the reference manual *CPLEX File Formats* for more detail and the *CPLEX User's Manual* for additional information about file formats.

```
public IloBool feasOpt(const IloConstraintArray cts, const IloNumArray prefs)
```

The method feasOpt computes a minimal relaxation of constraints in the active model in order to make the model feasible. On successful completion, the method installs a solution vector that is feasible for the minimum-cost relaxation. This solution can be queried with query methods, such as IloCplex::getValues or IloCplex::getInfeasibility.

The method feasOpt provides several different metrics for determining what constitutes a minimum relaxation. The metric is specified by the parameter FeasOptMode. The method feasOpt can also optionally perform a second optimization phase where the original objective is optimized, subject to the constraint that the associated relaxation metric must not exceed the relaxation value computed in the first phase.

The user may specify values (known as preferences) to express relative preferences for relaxing constraints. A larger preference specifies a greater willingness to relax the corresponding constraint. Internally, feasOpt uses the reciprocal of the preference to weight the relaxations of the associated bounds in the phase one cost function. A negative or 0 (zero) value as a preference specifies that the corresponding constraint must **not** be relaxed. If a preference is specified for a ranged constraint, that preference is used for both, its upper and lower bound. The preference for relaxing constraint cts[i] should be provided in prefs[i].

The array cts need not contain all constraints in the model. Only constraints directly added to the model can be specified. If a constraint is not present, it will not be relaxed.

IloAnd can be used to group constraints to be treated as one. Thus, according to the various Inf relaxation penalty metrics, all constraints in a group can be relaxed for a penalty of one unit. Similarly, according to the various Quad metrics, the penalty for relaxing a group grows as the square of the sum of the individual member relaxations, rather than as the sum of the squares of the individual relaxations.

If enough variables or constraints were allowed to be relaxed, the function will return IloTrue; otherwise, it returns IloFalse.

The active model is not changed by this method. If feasOpt finds a feasible solution, it returns the solution and the corresponding objective in terms of the original model.

The parameters CutUp, CutLo, ObjULim, ObjLLim do not influence this method. If you want to study infeasibilities introduced by those parameters, consider adding an objective function constraint to your model to enforce their effect before you invoke this method.

See Also: IloCplex::Relaxation

```
public IloBool feasOpt(const IloRangeArray rngs, const IloNumArray rnglb, const
IloNumArray rngub)
```

Attempts to find a minimum feasible relaxation of the active model by relaxing the bounds of the constraints specified in `rngs`. Preferences are specified in `rnglb` and `rngub` on input.

The parameters `CutUp`, `CutLo`, `ObjULim`, `ObjLLim` do not influence this method. If you want to study infeasibilities introduced by those parameters, consider adding an objective function constraint to your model to enforce their effect before you invoke this method.

The method returns `IloTrue` if it finds a feasible relaxation.

```
public IloBool feasOpt(const IloNumVarArray vars, const IloNumArray varlb, const
IloNumArray varub)
public IloBool feasOpt(const IloIntVarArray vars, const IloNumArray varlb, const
IloNumArray varub)
```

Attempts to find a minimum feasible relaxation of the active model by relaxing the bounds of the variables specified in `vars` as specified in `varlb` and `varub`.

The parameters `CutUp`, `CutLo`, `ObjULim`, `ObjLLim` do not influence this method. If you want to study infeasibilities introduced by those parameters, consider adding an objective function constraint to your model to enforce their effect before you invoke this method.

The method returns `IloTrue` if it finds a feasible relaxation.

```
public IloBool feasOpt(const IloRangeArray rngs, const IloNumArray rnglb, const
IloNumArray rngub, const IloNumVarArray vars, const IloNumArray varlb, const
IloNumArray varub)
public IloBool feasOpt(const IloRangeArray rngs, const IloNumArray rnglb, const
IloNumArray rngub, const IloIntVarArray vars, const IloNumArray varlb, const
IloNumArray varub)
```

The method `feasOpt` computes a minimal relaxation of the range and variable bounds of the active model in order to make the model feasible. On successful completion, the method installs a solution vector that is feasible for the minimum-cost relaxation. This solution can be queried with query methods, such as `IloCplex::getValues` or `IloCplex::getInfeasibility`.

The method `feasOpt` provides several different metrics for determining what constitutes a minimum relaxation. The metric is specified by the parameter `FeasOptMode`. The method `feasOpt` can also optionally perform a second optimization phase where the original objective is optimized, subject to the constraint that the associated relaxation metric must not exceed the relaxation value computed in the first phase.

The user may specify values (known as preferences) to express relative preferences for relaxing bounds. A larger preference specifies a greater willingness to relax the corresponding bound. Internally, `feasOpt` uses the reciprocal of the preference to weight the relaxations of the associated bounds in the phase one cost function. A negative or 0 (zero) value as a preference specifies that the corresponding bound must **not** be relaxed. The preference for relaxing the lower bound of constraint `rngs[i]` should be provided in `rnglb[i]`; and likewise the preference for relaxing the upper bound of constraint `rngs[i]` in `rngub[i]`. Similarly, the preference for relaxing the lower bound of variable `vars[i]` should be provided in `varlb[i]`, and the preference for relaxing its upper bound in `varub[i]`.

Arrays `rngs` and `vars` need not contain all ranges and variables in the model. If a range or variable is not present, its bounds are not relaxed. Only constraints directly added to the model can be specified.

If enough variables or constraints were allowed to be relaxed, the function will return `IloTrue`; otherwise, it returns `IloFalse`.

The active model is not changed by this method. If `feasOpt` finds a feasible solution, it returns the solution and the corresponding objective in terms of the original model.

The parameters CutUp, CutLo, ObjULim, ObjLLim do not influence this method. If you want to study infeasibilities introduced by those parameters, consider adding an objective function constraint to your model to enforce their effect before you invoke this method.

See Also: IloCplex::Relaxation

```
public void freePresolve()
```

This method frees the presolved problem. Under the default setting of parameter Reduce, the presolved problem is freed when an optimal solution is found; however, it is not freed if Reduce has been set to 1 (primal reductions) or to 2 (dual reductions). In these instances, the function freePresolve can be used when necessary to free it manually.

```
public IloCplex::Aborter getAborter()
```

Returns a handle to the aborter being used by the invoking IloCplex object.

```
public IloCplex::Algorithm getAlgorithm() const
```

This method returns the algorithm type that was used to solve the most recent model in cases where it was not a MIP.

```
public void getAX(IloNumArray val, const IloRangeArray con) const
```

Computes A times X, where A is the corresponding LP constraint matrix.

For the constraints in con, this method places the values of the expressions, or, equivalently, the activity levels of the constraints for the current solution of the invoking IloCplex object into the array val. Array val is resized to the same size as array con, and val[i] will contain the slack value for constraint con[i]. All ranges in con must be part of the extracted model.

```
public IloNum getAX(const IloRange range) const
```

Computes A times X, where A is the corresponding LP constraint matrix.

This method returns the value of the expression of the constraint range, or, equivalently, its activity level, for the current solution of the invoking IloCplex object. The range must be part of the extracted model.

```
public IloCplex::BasisStatus getBasisStatus(const IloConstraint con) const
```

This method returns the basis status of the implicit slack or artificial variable created for the constraint con.

```
public IloCplex::BasisStatus getBasisStatus(const IloIntVar var) const
```

This method returns the basis status for the variable var.

```
public IloCplex::BasisStatus getBasisStatus(const IloNumVar var) const
```

This method returns the basis status for the variable `var`.

```
public void getBasisStatuses(IloCplex::BasisStatusArray cstat, const IloNumVarArray
var, IloCplex::BasisStatusArray rstat, const IloConstraintArray con) const
public void getBasisStatuses(IloCplex::BasisStatusArray cstat, const IloIntVarArray
var, IloCplex::BasisStatusArray rstat, const IloConstraintArray con) const
```

This method puts the basis status of each variable in `var` into the corresponding element of the array `cstat`, and it puts the status of each row in `con` (an array of ranges or constraints) into the corresponding element of the array `rstat`. Arrays `rstat` and `cstat` are resized accordingly.

```
public void getBasisStatuses(IloCplex::BasisStatusArray stat, const
IloConstraintArray con) const
```

This method puts the basis status of each constraint in `con` into the corresponding element of the array `stat`. Array `stat` is resized accordingly.

```
public void getBasisStatuses(IloCplex::BasisStatusArray stat, const IloNumVarArray
var) const
public void getBasisStatuses(IloCplex::BasisStatusArray stat, const IloIntVarArray
var) const
```

This method puts the basis status of each variable in `var` into the corresponding element of the array `stat`. Array `stat` is resized accordingly.

```
public IloNum getBestObjValue() const
```

This method accesses the currently best known bound of all the remaining open nodes in a branch-and-cut tree.

It is computed for a minimization problem as the minimum objective function value of all remaining unexplored nodes. Similarly, it is computed for a maximization problem as the maximum objective function value of all remaining unexplored nodes.

For a regular MIP optimization, this value is also the best known bound on the optimal solution value of the MIP problem. In fact, when a problem has been solved to optimality, this value matches the optimal solution value.

However, for the method `populate`, the value can also exceed the optimal solution value if CPLEX has already solved the model to optimality but continues to search for additional solutions.

```
public void getBoundSA(IloNumArray lblower, IloNumArray lbupper, IloNumArray
ublower, IloNumArray ubupper, const IloNumVarArray vars) const
public void getBoundSA(IloNumArray lblower, IloNumArray lbupper, IloNumArray
ublower, IloNumArray ubupper, const IloIntVarArray vars) const
```

For the given set of variables `vars`, bound sensitivity information is computed. When the method returns, the element `lblower[j]` and `lbupper[j]` will contain the lowest and highest value the lower bound of variable `vars[j]` can assume without affecting the optimality of the solution. Likewise, `ublower[j]` and `ubupper[j]` will contain the lowest and highest value the upper bound of variable `vars[j]` can assume without affecting the optimality of the solution. The arrays `lblower`, `lbupper`, `ublower`, and `ubupper` will be resized to the size of array `vars`. The value 0 (zero) can be passed for any of the return arrays if the information is not desired.

```
public IloCplex::ConflictStatus getConflict(IloConstraint con) const
```

Returns the conflict status for the constraint `con`.

Possible return values are:

`IloCplex::ConflictMember` the constraint has been proven to participate in the conflict.

`IloCplex::ConflictPossibleMember` the constraint has not been proven **not** to participate in the conflict; in other words, it might participate, though it might not.

The constraint `con` must be one that has previously been passed to `refineConflict` including `IloAnd` constraints.

```
public IloCplex::ConflictStatusArray getConflict(IloConstraintArray cons) const
```

Returns the conflict status for each of the constraints specified in `cons`.

The element `i` is the conflict status for the constraint `cons[i]` and can take the following values:

`IloCplex::ConflictMember` the constraint has been proven to participate in the conflict.

`IloCplex::ConflictPossibleMember` the constraint has not been proven **not** to participate in the conflict; in other words, it might participate, though it might not.

The constraints passed in `cons` must be among the same ones that have previously been passed to `refineConflict`, including `IloAnd` constraints.

```
public IloCplex::CplexStatus getCplexStatus() const
```

This method returns the CPLEX status of the invoking algorithm. For possible CPLEX values, see the enumeration type `IloCplex::CplexStatus`.

See also the topic *Interpreting Solution Quality* in the *CPLEX User's Manual* for more information about a status associated with infeasibility or unboundedness.

```
public IloCplex::CplexStatus getCplexSubStatus() const
```

This method accesses the solution status of the last node problem that was solved in the event of an error termination in the previous invocation of `IloCplex::solve`. The method `IloCplex::getCplexSubStatus` returns `0` in the event of a normal termination. If the invoking `IloCplex` object is continuous, this is equivalent to the status returned by the method `IloCplex::getCplexStatus`.

```
public IloNum getCplexTime() const
```

This method returns a time stamp.

To measure elapsed time in seconds between a starting point and ending point of an operation, take the time stamp at the starting point; take the time stamp at the ending point; subtract the starting time stamp from the ending time stamp.

This computation measures either wall clock time (also known as real time) or CPU time, depending on the parameter `ClockType`.

The absolute value of the time stamp is not meaningful.

```
public IloNum getCutoff() const
```

This method returns the MIP cutoff value being used during the MIP optimization. In a minimization problem, all nodes are pruned that have an optimal solution value of the continuous relaxation that is larger than the current cutoff value. The cutoff is updated with the incumbent. If the invoking `IloCplex` object is an LP or QP, `+IloInfinity` or `-IloInfinity` is returned, depending on the optimization sense.

```
public IloBool getDefault(IloCplex::BoolParam parameter) const
public const char * getDefault(IloCplex::StringParam parameter) const
public IloNum getDefault(IloCplex::NumParam parameter) const
public IloInt getDefault(IloCplex::IntParam parameter) const
```

This method returns the default setting of the specified `parameter`.

```
public IloCplex::DeleteMode getDeleteMode() const
```

This method returns the current delete mode of the invoking `IloCplex` object.

```
public IloCplex::BranchDirection getDirection(IloNumVar var) const
public IloCplex::BranchDirection getDirection(IloIntVar var) const
```

This method returns the branch direction previously assigned to variable `var` with the method `IloCplex::setDirection` or `IloCplex::setDirections`. If no direction has been assigned, `IloCplex::BranchGlobal` will be returned.

```
public void getDirections(IloCplex::BranchDirectionArray dir, const IloNumVarArray
var) const
public void getDirections(IloCplex::BranchDirectionArray dir, const IloIntVarArray
var) const
```

This method returns the branch directions previously assigned to variables listed in `var` with the method `setDirection` or `setDirections`. When the function returns, `dir[i]` will contain the branch direction assigned for variables `var[i]`. If no branch direction has been assigned to `var[i]`, `dir[i]` will be set to `IloCplex::BranchGlobal`.

```
public IloExtractable getDiverging() const
```

This method returns the diverging variable or constraint, in a case where the primal Simplex algorithm has determined the problem to be infeasible. The returned extractable is either an `IloNumVar` or an `IloConstraint` object extracted to the invoking `IloCplex` optimizer; it is of type `IloNumVar` if the diverging column corresponds to a variable, or of type `IloConstraint` if the diverging column corresponds to the slack variable of a constraint.

```
public IloNum getDiversityFilterLowerCutoff(IloCplex::FilterIndex filter) const
```

Given the index of a diversity filter associated with the solution pool, this method returns the lower cutoff value of that filter.

```
public void getDiversityFilterRefVals(IloCplex::FilterIndex filter, IloNumArray)
const
```

Accesses the reference values declared in a diversity filter specified by its index in the solution pool.

```
public IloNum getDiversityFilterUpperCutoff(IloCplex::FilterIndex filter) const
```

Given the index of a diversity filter associated with the solution pool, this method returns the lower cutoff value of that filter.

```
public void getDiversityFilterWeights(IloCplex::FilterIndex filter, IloNumArray)
const
```

Accesses the weights declared in a diversity filter specified by its index in the solution pool.

```
public IloNum getDual(const IloRange range) const
```

This method returns the dual value associated with the constraint `range` in the current solution of the invoking algorithm.

```
public void getDuals(IloNumArray val, const IloRangeArray con) const
```

This method puts the dual values associated with the ranges in the array `con` into the array `val`. Array `val` is resized to the same size as array `con`, and `val[i]` will contain the dual value for constraint `con[i]`.

```
public IloCplex::FilterIndex getFilterIndex(const char * lname_str) const
```

Accesses the index of a filter specified by its name.

```
public IloCplex::FilterType getFilterType(IloCplex::FilterIndex filter) const
```

Given the index of a filter associated with the solution pool, this method returns the type of that filter.

```
public void getFilterVars(IloCplex::FilterIndex filter, IloNumVarArray) const
```

Accesses the variables of a diversity filter specified by its index.

```
public IloInt getIncumbentNode() const
```

This method returns the node number where the current incumbent was found. If the invoking `IloCplex` object is an LP or a QP, 0 (zero) is returned.

```
public IloNum getInfeasibilities(IloNumArray infeas, const IloIntVarArray var)
const
```

This method puts the infeasibility values of the integer variables in array `var` for the current solution into the array `infeas`. The infeasibility value is 0 (zero) if the variable bounds are satisfied. If the infeasibility value is negative, it specifies the amount by which the lower bound of the variable must be changed; if the value is positive, it specifies the amount by which the upper bound of the variable must be changed. This method does not check for integer infeasibility. The array `infeas` is automatically resized to the same length as array `var`, and `infeas[i]` will contain the infeasibility value for variable `var[i]`. This method returns the maximum absolute infeasibility value over all integer variables in `var`.

```
public IloNum getInfeasibilities(IloNumArray infeas, const IloNumVarArray var)
const
```

This method puts the infeasibility values of the numeric variables in array `var` for the current solution into the array `infeas`. The infeasibility value is 0 (zero) if the variable bounds are satisfied. If the infeasibility value is negative, it specifies the amount by which the lower bound of the variable must be changed; if the value is positive, it specifies the amount by which the upper bound of the variable must be changed. The array `infeas` is automatically resized to the same length as array `var`, and `infeas[i]` will contain the infeasibility value for variable `var[i]`. This method returns the maximum absolute infeasibility value over all numeric variables in `var`.

```
public IloNum getInfeasibilities(IloNumArray infeas, const IloConstraintArray con)
const
```

This method puts the infeasibility values of the current solution for the constraints specified by the array `con` into the array `infeas`. The infeasibility value is 0 (zero) if the constraint is satisfied. More specifically, for a range with finite lower bound and upper bound, if the infeasibility value is negative, it specifies the amount by which the lower bound of the range must be changed; if the value is positive, it specifies the amount by which the upper bound of the range must be changed. For a more general constraint such as `IloOr`, `IloAnd`, `IloSOS1`, or `IloSOS2`, the infeasibility value returned is the maximal absolute infeasibility value over all range constraints and variables created by the extraction of the queried constraint. Array `infeas` is resized to the same size as array `range`, and `infeas[i]` will contain the infeasibility value for constraint `range[i]`. This method returns the maximum absolute infeasibility value over all constraints in `con`.

```
public IloNum getInfeasibility(const IloIntVar var) const
```

This method returns the infeasibility of the integer variable `var` in the current solution. The infeasibility value returned is 0 (zero) if the variable bounds are satisfied. If the infeasibility value is negative, it specifies the amount by which the lower bound of the variable must be changed; if the value is positive, it specifies the amount by which the upper bound of the variable must be changed. This method does not check for integer infeasibility.

```
public IloNum getInfeasibility(const IloNumVar var) const
```

This method returns the infeasibility of the numeric variable `var` in the current solution. The infeasibility value returned is 0 (zero) if the variable bounds are satisfied. If the infeasibility value is negative, it specifies the amount by which the lower bound of the variable must be changed; if the value is positive, it specifies the amount by which the upper bound of the variable must be changed.

```
public IloNum getInfeasibility(const IloConstraint con) const
```

This method returns the infeasibility of the current solution for the constraint `code`. The infeasibility value is 0 (zero) if the constraint is satisfied. More specifically, for a range with finite lower bound and upper bound, if the infeasibility value is negative, it specifies the amount by which the lower bound of the range must be changed; if the value is positive, it specifies the amount by which the upper bound of the range must be changed. For a more general constraint such as `IloOr`, `IloAnd`, `IloSOS1`, or `IloSOS2`, the infeasibility value returned is the maximal absolute infeasibility value over all range constraints and variables created by the extraction of the queried constraint.

```
public IloInt getMax(IloCplex::IntParam parameter) const
public IloNum getMax(IloCplex::NumParam parameter) const
```

These method return the maximum allowed value for the parameter `parameter`.

```
public IloInt getMin(IloCplex::IntParam parameter) const
public IloNum getMin(IloCplex::NumParam parameter) const
```

These method return the minimum allowed value for the parameter `parameter`.

```
public IloNum getMIPRelativeGap() const
```

This method accesses the relative objective gap for a MIP optimization.

For a **minimization** problem, this value is computed by

```
(bestinteger - bestobjective) / (1e-10 + |bestobjective|)
```

where `bestinteger` is the value returned by `IloCplex::getObjValue` and `bestobjective` is the value returned by `IloCplex::getBestObjValue`. For a **maximization** problem, the value is computed by:

```
(bestobjective - bestinteger) / (1e-10 + |bestobjective|)
```

```
public IloCplex::MIPStartEffort getMIPStart(IloInt mipstartindex, const
IloNumVarArray vars, IloNumArray vals, IloBoolArray isset)
```

Returns the level of effort associated with the MIP start identified by `mipstartindex` and defined by the arrays `vars` and `vals`. The argument `isset` designates which members of `vars` and which corresponding values of `vals` participate in the MIP start.

```
public IloInt getMIPStartIndex(const char * lname_str) const
```

Returns the index of the MIP start designated by its name.

```
public const char * getMIPStartName(IloInt mipstartindex)
```

Returns the name of the MIP start identified by its index.

```
public IloInt getNbarrierIterations() const
```

This method returns the number of barrier iterations from the last solve.

```
public IloInt getNbinVars() const
```

This method returns the number of binary variables in the matrix representation of the active model in the invoking `IloCplex` object.

```
public IloInt getNcols() const
```

This method returns the number of columns extracted for the invoking algorithm. There may be differences in the number returned by this function and the number of object of type `IloNumVar` and its subclasses in the model that is extracted. This is because automatic transformation of nonlinear constraints and expressions may introduce new variables.

```
public IloInt getNcrossDExch() const
```

This method returns the number of dual exchange operations in the crossover of the last call to method `solve` or `solveFixed`, if barrier with crossover has been used for solving an LP or QP.

```
public IloInt getNcrossDPush() const
```

This method returns the number of dual push operations in the crossover of the last call to `IloCplex::solve` or `IloCplex::solveFixed`, if barrier with crossover was used for solving an LP or QP.

```
public IloInt getNcrossPExch() const
```

This method returns the number of primal exchange operations in the crossover of the last call of method `IloCplex::solve` or `IloCplex::solveFixed`, if barrier with crossover was used for solving an LP of QP.

```
public IloInt getNcrossPPush() const
```

This method returns the number of primal push operations in the crossover of the last call of method `IloCplex::solve` or `IloCplex::solveFixed`, if barrier with crossover was used for solving an LP or QP.

```
public IloInt getNcuts(IloCplex::CutType which) const
```

This method returns the number of cuts of the specified type in use at the end of the previous mixed integer optimization. If the invoking `IloCplex` object is not a MIP, it returns 0.

```
public IloInt getNdualSuperbasics() const
```

This method returns the number of dual superbasic variables in the current solution of the invoking `IloCplex` object.

```
public IloInt getNfilters() const
```

Returns the number of filters currently associated with the solution pool.

```
public IloInt getNintVars() const
```

This method returns the number of integer variables in the matrix representation of the active model in the invoking `IloCplex` object.

```
public IloInt getNiterations() const
```

This method returns the number of iterations that occurred during the last call to the method `IloCplex::solve` in the invoking algorithm.

```
public int getNMIPStarts() const
```

Returns the number of MIP starts associated with the current problem.

```
public IloInt getNnodes() const
```

This method returns the number of branch-and-cut nodes that were processed in the current solution. If the invoking `IloCplex` object is not a MIP, it returns 0.

```
public IloInt getNnodesLeft() const
```

This method returns the number of branch-and-cut nodes that remain to be processed in the current solution. If the invoking `IloCplex` object is not a MIP, it returns 0.

```
public IloInt getNNZs() const
```

This method returns the number of nonzeros extracted to the constraint matrix A of the invoking algorithm.

```
public IloInt getNphaseOneIterations() const
```

If a simplex method was used for solving continuous model, this method returns the number of iterations in phase 1 of the last call to `IloCplex::solve` or `IloCplex::solveFixed`.

```
public IloInt getNprimalSuperbasics() const
```

This method returns the number of primal superbasic variables in the current solution of the invoking `IloCplex` object.

```
public IloInt getNQCs() const
```

This method returns the number of quadratic constraints extracted from the active model for the invoking algorithm. This number may be different from the total number of constraints in the active model because linear constraints are not accounted for in this function.

See Also: IloCplex::getNrows

```
public IloInt getNrows() const
```

This method returns the number of rows extracted for the invoking algorithm. There may be differences in the number returned by this function and the number of `IloRange`s and `IloConstraint`s in the model that is extracted. This is because quadratic constraints are not accounted for by method `getNrows` and automatic transformation of nonlinear constraints and expressions may introduce new constraints.

See Also: IloCplex::getNQCs

```
public IloInt getNsemiContVars() const
```

This method returns the number of semicontinuous variables in the matrix representation of the active model in the invoking `IloCplex` object.

```
public IloInt getNsemiIntVars() const
```

This method returns the number of semi-integer variables in the matrix representation of the active model in the invoking `IloCplex` object.

```
public IloInt getNsiftingIterations() const
```

This method returns the number of sifting iterations performed for solving the last LP with algorithm type `IloCplex::Sifting`, or, equivalently, the number of work LPs that have been solved for it.

```
public IloInt getNsiftingPhaseOneIterations() const
```

This method returns the number of sifting iterations performed for solving the last LP with algorithm type `IloCplex::Sifting` in order to achieve primal feasibility.

```
public IloInt getNSOSs() const
```

This method returns the number of SOSs extracted for the invoking algorithm. There may be differences in the number returned by this function and the number of numeric variables (that is, instances of the class `IloNumVar`, and so forth) in the model that is extracted because piecewise linear functions are extracted as a set of SOSs.

```
public IloObjective getObjective() const
```

This method returns the instance of `IloObjective` that has been extracted to the invoking instance of `IloCplex`. If no objective has been extracted, an empty handle is returned.

If you need only the current value of the objective, for example to use in a callback, consider one of these methods instead:

- `ContinuousCallbackI::getObjValue`
- `ControlCallbackI::getObjValue`
- `GoalI::getObjValue`
- `IncumbentCallbackI::getObjValue`
- `NetworkCallbackI::getObjValue`
- `NodeCallbackI::getObjValue`
- `NodeEvaluatorI::getObjValue`

```
public void getObjSA(IloNumArray lower, IloNumArray upper, const IloNumVarArray
vars) const
public void getObjSA(IloNumArray lower, IloNumArray upper, const IloIntVarArray
cols) const
```

This method performs objecitve sensitivity analysis for the variables specified in array `vars`. When this method returns `lower[i]` and `upper[i]` will contain the lowest and highest value the objective function coefficient for variable `vars[i]` can assume without affecting the optimality of the solution. The arrays `lower` and `upper` will be resized to the size of array `vars`. If any of the information is not requested, 0 (zero) can be passed for the corresponding array parameter.

```
public IloNum getObjValue(IloInt soln) const
```

This member function returns the numeric value of the objective function for the solution pool member indexed by `soln`. The `soln` argument may be omitted or given a value of -1 in order to access the current solution.

```
public IloBool getParam(IloCplex::BoolParam parameter) const
public const char * getParam(IloCplex::StringParam parameter) const
public IloNum getParam(IloCplex::NumParam parameter) const
public IloInt getParam(IloCplex::IntParam parameter) const
```

This method returns the current setting of `parameter` in the invoking algorithm.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

```
public IloCplex::ParameterSet getParameterSet()
```

Returns a parameter set corresponding to the present parameter state.

If the method fails, an exception of type `IloException`, or one of its derived classes, is thrown.

Returns:

The parameter set.

```
public void getPriorities(IloNumArray pri, const IloNumVarArray var) const
public void getPriorities(IloNumArray pri, const IloIntVarArray var) const
```

This method returns query branch priorities previously assigned to variables listed in `var` with the method `setPriority` or `setPriorities`. When the function returns, `pri[i]` will contain the priority value assigned for variables `var[i]`. If no priority has been assigned to `var[i]`, `pri[i]` will contain 0 (zero).

```
public IloNum getPriority(IloNumVar var) const
public IloNum getPriority(IloIntVar var) const
```

This method returns the priority previously assigned to the variable `var` with the method `setPriority` or `setPriorities`. It returns 0 (zero) if no priority has been assigned.

```
public IloNum getQuality(IloCplex::Quality q, IloInt soln, IloConstraint * rng,
IloNumVar * var=0) const
public IloNum getQuality(IloCplex::Quality q, IloInt soln, IloNumVar * var=0,
IloConstraint * rng=0) const
```

These methods return the requested quality measure for a member of the solution pool. The `soln` argument may be given a value of -1 to return the quality meausre for the current solution.

Some quality measures are related to a variable or a constraint. For example, `IloCplex::MaxPrimalInfeas` is related to the variable or constraint where the maximum infeasibility (bound violation) occurs. If this information is also requested, pointers to instances of `IloNumVar` or `IloConstraint` may be passed as optional arguments specifying where the relevant variable or range will be written. However, if the constraint has been implicitly created (for example, because of automatic linearization), a null handle will be returned for these arguments.

```
public IloNum getQuality(IloCplex::Quality q, IloNumVar * var=0, IloConstraint *
rng=0) const
public IloNum getQuality(IloCplex::Quality q, IloConstraint * rng, IloNumVar *
var=0) const
```

These methods return the requested quality measure.

Some quality measures are related to a variable or a constraint. For example, `IloCplex::MaxPrimalInfeas` is related to the variable or constraint where the maximum infeasibility (bound violation) occurs. If this information is also requested, pointers to instances of `IloNumVar` or `IloConstraint` may be passed as optional arguments specifying where the relevant variable or range will be written. However, if the constraint has been implicitly created (for example, because of automatic linearization), a null handle will be returned for these arguments.

```
public void getRangeFilterCoefs(IloCplex::FilterIndex filter, IloNumArray) const
```

Accesses the coefficients (that is, the weights) declared in the range filter specified by its index.

```
public IloNum getRangeFilterLowerBound(IloCplex::FilterIndex filter) const
```

Given the index of a range filter associated with the solution pool, this method returns the lower bound of that filter.

```
public IloNum getRangeFilterUpperBound(IloCplex::FilterIndex filter) const
```

Given the index of a range filter associated with the solution pool, this method returns the upper bound of that filter.

```
public void getRangeSA(IloNumArray lblower, IloNumArray lbupper, IloNumArray
ublower, IloNumArray ubupper, const IloRangeArray con) const
```

This method performs sensistivity analysis for the upper and lower bounds of the ranged constraints passed in the array `con`. When the method returns, `lblower[i]` and `lbupper[i]` will contain, respecitively, the lowest and the highest value that the lower bound of constraint `con[i]` can assume without affecting the optimality of the solution. Similarly, `ublower[i]` and `ubupper[i]` will contain, respectively, the lowest and the highest value that the upper bound of the constraint `con[i]` can assume without affecting the optimality of the solution. The arrays `lblower`, `lbupper`, `ublower`, and `ubupper` will be resized to the size of array `con`. If any of the information is not requested, `0` can be passed for the corresponding array parameter.

```
public void getRay(IloNumArray vals, IloNumVarArray vars) const
```

This method returns an unbounded direction (also known as a ray) corresponding to the present basis for an LP that has been determined to be an unbounded problem. CPLEX puts the the nonzero values of the unbounded direction into the array `vals`, and it puts the corresponding variables of the extracted model into the array `vars`.

Note

CPLEX modifies these arrays, resizing them if necessary. Any previous values in them will be overwritten.

```
public IloNum getReducedCost(const IloIntVar var) const
```

This method returns the reduced cost associated with `var` in the invoking algorithm.

```
public IloNum getReducedCost(const IloNumVar var) const
```

This method returns the reduced cost associated with `var` in the invoking algorithm.

```
public void getReducedCosts(IloNumArray val, const IloIntVarArray var) const
```

This method puts the reduced costs associated with the numeric variables of the array `var` into the array `val`. The array `val` is automatically resized to the same length as array `var`, and `val[i]` will contain the reduced cost for variable `var[i]`.

```
public void getReducedCosts(IloNumArray val, const IloNumVarArray var) const
```

This method puts the reduced costs associated with the variables in the array `var` into the array `val`. Array `val` is resized to the same size as array `var`, and `val[i]` will contain the reduced cost for variable `var[i]`.

```
public void getRHSSA(IloNumArray lower, IloNumArray upper, const IloRangeArray
cons) const
```

This method performs righthand side sensitivity analysis for the constraints specified in array `cons`. The

constraints must be of the form `cons[i]: expr[i] rel rhs[i]`. When this method returns `lower[i]` and `upper[i]` will contain the lowest and highest value `rhs[i]` can assume without affecting the optimality of the solution. The arrays `lower` and `upper` will be resized to the size of array `cons`. If any of the information is not requested, 0 (zero) can be passed for the corresponding array parameter.

```
public IloNum getSlack(const IloRange range, IloInt soln=IncumbentId) const
```

This method returns the slack value associated with the constraint `range` for a solution of the invoking algorithm. For a range with finite lower and upper bounds, the slack value consists of the difference between the expression of the range and its lower bound. The current solution is used if the `soln` argument is omitted or given the value -1; otherwise, the solution pool member indexed by `soln` is used.

```
public void getSlacks(IloNumArray val, const IloRangeArray con, IloInt
soln=IncumbentId) const
```

This method puts the slack values associated with the constraints specified by the array `con` into the array `val`. For a ranged constraint with finite lower and upper bounds, the slack value consists of the difference between the expression in the range and its lower bound. The current solution is used if the `soln` argument is omitted or given the value -1; otherwise, the solution pool member indexed by `soln` is used. Array `val` is resized to the same size as array `con`, and `val[i]` will contain the slack value for constraint `con[i]`.

```
public IloNum getSolnPoolMeanObjValue() const
```

Computes the mean of the objective values of the solutions currently in the solution pool.

```
public IloInt getSolnPoolNreplaced() const
```

Accesses the number of solutions that have been replaced according to the solution pool replacement strategy.

```
public IloInt getSolnPoolNsolns() const
```

Accesses the number of solutions currently in the solution pool.

```
public IloAlgorithm::Status getStatus() const
```

This method returns the status of the invoking algorithm.

For its CPLEX status, see the method `IloCplex::getCplexStatus`.

See also the topic *Interpreting Solution Quality* in the *CPLEX User's Manual* for more information about a status associated with infeasibility or unboundedness.

```
public IloCplex::Algorithm getSubAlgorithm() const
```

This method returns the type of the algorithm type that was used to solve most recent node of a MIP in the case of termination because of an error during mixed integer optimization.

```
public IloNum getValue(const IloObjective ob, IloInt soln) const
```

This method returns the value of the objective using the solution pool member indexed by `soln`. The `soln` argument may be omitted or given a value of -1 in order to access the current solution.

```
public IloNum getValue(const IloNumExprArg expr, IloInt soln) const
```

This method returns the value of the expression using the solution pool member indexed by `soln`. The `soln` argument may be omitted or given a value of -1 in order to access the current solution.

```
public IloNum getValue(const IloIntVar var, IloInt soln) const
```

This method returns the value from the solution pool member indexed by `soln` for the integer variable specified by `var`. The `soln` argument may be omitted or given a value of -1 in order to access the current solution.

```
public IloNum getValue(const IloNumVar var, IloInt soln) const
```

This method returns the value from the solution pool member indexed by `soln` for the numeric variable specified by `var`. The `soln` argument may be omitted or given a value of -1 in order to access the current solution.

```
public void getValues(const IloIntVarArray var, IloNumArray val, IloInt soln) const
```

This method puts the values from the solution pool member indexed by `soln` for the integer variables specified by the array `var` into the array `val`. The `soln` argument may be omitted or given a value of -1 in order to access the current solution. Array `val` is resized to the same size as array `var`, and `val[i]` will contain the solution value for variable `var[i]`.

```
public void getValues(IloNumArray val, const IloIntVarArray var, IloInt soln) const
```

This method puts the values from the solution pool member indexed by `soln` for the numeric variables specified by the array `var` into the array `val`. The `soln` argument may be omitted or given a value of -1 in order to access the current solution. Array `val` is resized to the same size as array `var`, and `val[i]` will contain the solution value for variable `var[i]`.

```
public void getValues(IloNumArray val, const IloNumVarArray var, IloInt soln) const
```

This method puts the values from the solution pool member indexed by `soln` for the numeric variables specified by the array `var` into the array `val`. The `soln` argument may be omitted or given a value of -1 in order to access the current solution. Array `val` is resized to the same size as array `var`, and `val[i]` will contain the solution value for variable `var[i]`.

```
public void getValues(const IloIntVarArray var, IloNumArray val) const
```

This method puts the solution values of the integer variables specified by the array `var` into the array `val`. Array `val` is resized to the same size as array `var`, and `val[i]` will contain the solution value for variable `var[i]`.

```
public void getValues(IloNumArray val, const IloIntVarArray var) const
```

This method puts the solution values of the integer variables specified by the array `var` into the array `val`. Array `val` is resized to the same size as array `var`, and `val[i]` will contain the solution value for variable `var[i]`.

```
public void getValues(const IloNumVarArray var, IloNumArray val) const
```

This member function accepts an array of variables `vars` and puts the corresponding values into the array `vals`; the corresponding values come from the current solution of the invoking algorithm. The array `vals` must be a clean, empty array when you pass it to this member function.

If there are no values to return for `vars`, this member function raises an error. On platforms that support C++ exceptions, when exceptions are enabled, this member function throws the exception `NotExtractedException` in such a case.

```
public void getValues(IloNumArray val, const IloNumVarArray var) const
```

This method puts the solution values of the numeric variables specified by the array `var` into the array `val`. Array `val` is resized to the same size as array `var`, and `val[i]` will contain the solution value for variable `var[i]`.

```
public const char * getVersion() const
```

This method returns a string specifying the version of `IloCplex`.

```
public void importModel(IloModel & m, const char * filename) const
```

This method reads a model from the file specified by `filename` into `model`. Typically `model` will be an empty, unextracted model on entry to this method. The invoking `IloCplex` object is not affected when you call this method unless `model` is its extracted model; follow this method with a call to `IloCplex::extract` in order to extract the imported model to the invoking `IloCplex` object.

When this methods reads a file, new modeling objects, as required by the input file, are created and added to any existing modeling objects in the `model` passed as an argument. Note that any previous modeling objects in `model` are not removed; precede the call to `importModel` with explicit calls to `IloModel::remove` if you need to remove them.

```
public void importModel(IloModel & m, const char * filename, IloObjective & obj,
IloNumVarArray vars, IloRangeArray rngs, IloRangeArray lazy=0, IloRangeArray
cuts=0) const
```

This method is a simplification of the method `importModel` that does not provide arrays to return SOSs. This method is easier to use in the case where you are dealing with continuous models because in such a case you already know that no SOS will be present.

```
public void importModel(IloModel & model, const char * filename, IloObjective &
obj, IloNumVarArray vars, IloRangeArray rngs, IloSOS1Array sos1, IloSOS2Array sos2,
IloRangeArray lazy=0, IloRangeArray cuts=0) const
```

This method reads a model from the file specified by `filename` into `model`. Typically `model` will be an empty, unextracted model on entry to this method. The invoking `IloCplex` object is not affected when you call this method unless `model` is its extracted model; follow this method with a call to `IloCplex::extract` in order to extract the imported model to the invoking `IloCplex` object.

When this methods reads a file, new modeling objects, as required by the input file, are created and added to any existing modeling objects in the `model` passed as an argument. Note that any previous modeling objects in `model` are not removed; precede the call to `importModel` with explicit calls to `IloModel::remove` if you need to remove them.

As this method reads a model from a file, it places the objective it has read in `obj`, the variables it has read in the array `vars`, * the ranges it has read in the array `rngs`; and the Special Ordered Sets (SOS) it has read in the arrays `sos1` and `sos2`.

Note

CPLEX resizes these arrays for you to accomodate the returned objects.

Note

This note is for advanced users only. The two arrays `lazy` and `cuts` are filled with the lazy constraints and user cuts that may be included in the model in the file `filename`.

The format of the file is determined by the extension of the file name. The following extensions are recognized on most platforms:

- `.sav`
- `.mps`
- `.lp`
- `.sav.gz` (if gzip is properly installed)
- `.mps.gz` (if gzip is properly installed)
- `.lp.gz` (if gzip is properly installed)

Microsoft Windows does not support gzipped files for this API.

```
public IloBool isDualFeasible() const
```

This method returns `IloTrue` if a dual feasible solution is recorded in the invoking `IloCplex` object and can be queried.

```
public IloBool isMIP() const
```

This method returns `IloTrue` if the invoking algorithm has extracted a model that is a MIP (mixed-integer programming problem) and `IloFalse` otherwise. Member functions for accessing duals and reduced cost basis work only if the model is not a MIP.

```
public IloBool isPrimalFeasible() const
```

This method returns `IloTrue` if a primal feasible solution is recorded in the invoking `IloCplex` object and can be queried.

```
public IloBool isQC() const
```

This method returns `IloTrue` if the invoking algorithm has extracted a model that is quadratically constrained. Otherwise, it returns `IloFalse`. For an explanation of *quadratically constrained* see the topic QCP in the *CPLEX User's Manual*.

```
public IloBool isQO() const
```

This method returns `IloTrue` if the invoking algorithm has extracted a model that has quadratic objective function terms. Otherwise, it returns `IloFalse`.

```
public static IloCplex::Goal LimitSearch(IloCplex cplex, IloCplex::Goal goal,
IloCplex::SearchLimit limit)
```

This method creates and returns a goal that puts the search specified by `goal` under the limit defined by `limit`. Only the subtree controlled by `goal` will be subjected to limit `limit`.

```
public IloBool populate()
```

The method `populate` generates multiple solutions to a mixed integer programming (MIP) model. In other words, it populates the solution pool of the model currently extracted by the invoking `IloCplex` object. Like the method `solve`, this method returns `IloTrue` if it finds a solution (not necessarily an optimal solution).

The algorithm that populates the solution pool works in two phases:

In the first phase, it solves the model to optimality (or some stopping criterion set by the user) while it sets up a branch and cut tree for the second phase.

In the second phase, it generates multiple solutions by using the information computed and stored in the first phase and by continuing to explore the tree.

The amount of preparation in the first phase and the intensity of exploration in the second phase are controlled by the solution pool intensity parameter `SolnPoolIntensity`.

Optimality is not a stopping criterion for the `populate` method. Even if the optimality gap is zero, this method will still try to find alternative solutions. The **stopping criteria** for `populate` are these:

- Popoulate limit `PopulateLim`. This parameter controls how many solutions are generated before the method stops. Its default value is 20.
- Time limit `TiLim`, as in standard MIP optimization.
- Node limit `NodeLim`, as in standard MIP optimization.
- In the absence of other stopping criteria, `populate` stops when it cannot enumerate any more solutions. In particular, if the user specifies an objective tolerance with the relative or absolute solution pool gap parameters, `populate` stops if it cannot enumerate any more solutions within the specified objective tolerance. There may exist additional solutions that satisfy the specified objective tolerance; depending on the solution pool intensity parameter, `populate` may or may not enumerate all of them; according to certain settings of the solution pool intensity parameter, `populate` may stop when it has enumerated a subset of additional solutions satisfying the specified objective tolerance.

Successive calls to `populate` create solutions that are stored in the solution pool. However, each call to `populate` applies only to the subset of solutions created in the current call; the call does not affect the solutions already in the pool. In other words, solutions in the pool are persistent.

The user may call this routine independently of any MIP optimization of a model. In that case, it carries out the first and second phase itself.

The user may also call `populate` after standard MIP optimization. In the general case, the user reads the model, calls MIP optimization, then calls `populate`. The activity of MIP optimization constitutes the first phase of

the populate algorithm; `populate` then re-uses the information computed and stored by MIP optimization and thus carries out only the second phase.

The method `populate` does not try to generate multiple solutions for unbounded MIP models. As soon as the proof of unboundedness is obtained, `populate` stops.

```
public void presolve(IloCplex::Algorithm alg)
```

This method performs Presolve on the model. The enumeration `alg` tells Presolve which algorithm is intended to be used on the reduced model; `NoAlg` should be specified for MIP models.

```
public void protectVariables(const IloIntVarArray var)
```

Note

This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method specifies a set of integer variables that should not be substituted out of the problem. If presolve can fix a variable to a value, it is removed, even if it is specified in the protected list.

```
public void protectVariables(const IloNumVarArray var)
```

Note

This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method specifies a set of numeric variables that should not be substituted out of the problem. If presolve can fix a variable to a value, it is removed, even if it is specified in the protected list.

```
public void qpIndefCertificate(IloNumVarArray var, IloNumArray x)
public void qpIndefCertificate(IloIntVarArray var, IloNumArray x)
```

The quadratic objective terms in a QP model must form a positive semi-definite Q matrix (negative semi-definite for maximization). If `IloCplex` finds that this is not true, it will discontinue the optimization. In such cases, the `qpIndefCertificate` method can be used to compute assignments (returned in array `x`) to all variables (returned in array `var`) such that the quadratic term of the objective function evaluates to a negative value (`x'Q x < 0` in matrix terms) to prove the indefiniteness.

Note

CPLEX resizes these arrays for you.

```
public void readBasis(const char * name) const
```

Reads a simplex basis from the BAS file specified by `name`, and copies that basis into the invoking `IloCplex` object. The parameter `AdvInd` must be set to a nonzero value (e.g. its default setting) for the simplex basis to be used to start a subsequent optimization with one of the Simplex algorithms.

By convention, the file extension is `.bas`. The BAS file format is documented in the reference manual *CPLEX File Formats*.

```
public IloCplex::FilterIndexArray readFilters(const char * name)
```

Reads solution pool filters from a file in FLT format and copies the filters into an instance of `IloCplex`. This format is documented in the reference manual *CPLEX File Formats*.

```
public void readMIPStart(const char * name) const
```

This method has been **deprecated**. Use the method `IloCplex::readMIPStarts` instead.

```
public void readMIPStarts(const char * name) const
```

This method reads the MST file denoted by `name` and copies the MIP start information into the invoking `IloCplex` object. The parameter `AdvInd` must be turned on (its default: 1 (one)) in order for the MIP start information to be used to with a subsequent MIP optimization.

By convention, the file extension is `.mst`. The MST file format is documented in the reference manual *CPLEX File Formats* and in the stylesheet `solution.xsl` and schema `solution.xsd` in the `include` directory of the product. Examples of its use appear in the examples distributed with the product and in the *CPLEX User's Manual*.

```
public void readOrder(const char * filename) const
```

This method reads a priority order from a file in ORD format into the invoking `IloCplex` object. The names in the ORD file must match the names in the active model. The priority order will be associated with the model. The parameter `MipOrdInd` must be nonzero for the next invocation of the method `IloCplex::solve` to take the order into account.

By convention, the file extension is `.ord`. The ORD file format is documented in the reference manual *CPLEX File Formats*.

```
public void readParam(const char * name) const
```

Reads parameters and their settings from the file specified by `name` and applies them to the invoking `IloCplex` object. Parameters not listed in the parameter file will be reset to their default setting.

By convention, the file extension is `.prm`. The PRM file format is documented in the reference manual *CPLEX File Formats*.

```
public void readSolution(const char * name) const
```

Reads a solution from the SOL file denoted by `name` and copies this information into a CPLEX problem object. This routine is able to initiate a crossover from the barrier solution, to restart the simplex method with an advanced basis or to specify variable values for a MIP start. The parameter `AdvInd` must set to a nonzero value

(such as its default setting: 1 (one)) in order for the solution file to take effect with the method `solve`.

By convention, the file extension is `.sol`. The SOL file format is documented in the reference manual *CPLEX File Formats* and in the stylesheet `solution.xsl` and schema `solution.xsd` in the `include` directory of the product. Examples of its use appear in the examples distributed with the product and in the *CPLEX User's Manual*.

```
public IloBool refineConflict(IloConstraintArray cons, IloNumArray prefs)
```

The method `refineConflict` identifies a minimal conflict for the infeasibility of the current model or for a subset of the constraints of the current model. Since the conflict is minimal, removal of any one of these constraints will remove that particular cause for infeasibility. There may be other conflicts in the model; consequently, repair of a given conflict does not guarantee feasibility of the remaining model.

The constraints among which to look for a conflict are passed to this method through the argument `cons`. Only constraints directly added to the model can be specified.

Constraints may also be grouped by `IloAnd`. If any constraint in a group participates in the conflict, the entire group is determined to do so. No further detail about the constraints within that group is returned.

Groups or constraints may be assigned preference. A group or constraint with a higher preference is more likely to be included in the conflict. However, no guarantee is made when a minimal conflict is returned that other conflicts containing groups or constraints with higher preference do not exist.

To check whether the bounds of a variable cause a conflict, use an instance of the class `IloBound` to specify the lower and upper bounds of the variable in question. Use those bounds like constraints among the arguments you pass to `refineConflict`.

When this method returns, the conflict can be queried with the methods `getConflict`. The method `writeConflict` can write a file in LP format containing the conflict.

The parameters `CutUp`, `CutLo`, `ObjULim`, `ObjLLim` do not influence this method. If you want to study infeasibilities introduced by those parameters, consider adding an objective function constraint to your model to enforce their effect before you invoke this method.

Parameters:

cons An array of constraints among which to look for a conflict. The constraints may be any constraint in the active model, or a group of constraints organized by `IloAnd`. If a constraint does not appear in this array, the constraint is assigned a preference of 0 (zero). Thus such constraints are included in the conflict without any analysis. Only constraints directly added to the model can be specified.

prefs An array containing the preferences for the groups or constraints. `prefs[i]` specifies the preference for group or constraint `cons[i]`. A negative value specifies that the corresponding group or constraint should not be considered in the computation of a conflict. In other words, such groups are not considered part of the model. Groups with a preference of 0 (zero) are always considered to be part of the conflict. No further checking is performed on such groups.

Returns:

Boolean value reporting whether or not a conflict has been found.

```
public IloBool refineMIPStartConflict(IloInt mipstartindex, IloConstraintArray
cons, IloNumArray prefs)
```

This method accepts a MIP start, designated by its index, and identifies a minimal conflict for the infeasibility of that MIP start or for a subset of the constraints of that MIP start.

The parameters `CutUp`, `CutLo`, `ObjULim`, `ObjLLim` do not influence this method. If you want to study infeasibilities introduced by those parameters, consider adding an objective function as a constraint to your model to enforce their effect before you invoke this method.

When the MIP start was added to the current model, an effort level may have been associated with it to specify to CPLEX how much effort to expend in transforming the MIP start into a feasible solution. This method respects effort levels **except** level 1 (one): check feasibility. It does not check feasibility. Instead, CPLEX increases the effort level to 2 in order to solve the fixed model.

When this method returns, you can query the conflict with the method `IloCplex::getConflict` and write the conflict to a file in LP format with the method `IloCplex::writeConflict`.

Parameters:

cons	An array of constraints among which to look for a conflict. The constraints may be any constraint in the MIP start, or a group of constraints organized by `IloAnd`. If a constraint does not appear in this array, the constraint is assigned a preference of 0 (zero). Thus such constraints are included in the conflict without any analysis. Only constraints directly added to the model can be specified.
prefs	An array containing the preferences for the groups or constraints. `prefs[i]` specifies the preference for group or constraint `cons[i]`. A negative value specifies that the corresponding group or constraint should not be considered in the computation of a conflict. In other words, such groups are not considered part of the model. Groups with a preference of 0 (zero) are always considered to be part of the conflict. No further checking is performed on such groups.

```
public void remove(IloCplex::Aborter abort)
```

This method removes the aborter object `abort` from the invoking `IloCplex` object.

```
public void setBasisStatuses(const IloCplex::BasisStatusArray cstat, const
IloNumVarArray var, const IloCplex::BasisStatusArray rstat, const
IloConstraintArray con)
public void setBasisStatuses(const IloCplex::BasisStatusArray cstat, const
IloIntVarArray var, const IloCplex::BasisStatusArray rstat, const
IloConstraintArray con)
```

This method uses the array `cstats` to set the basis status of the variables in the array `var`; it uses the array `rstats` to set the basis status of the ranges in the array `con`.

```
public void setDefaults()
```

This method resets all CPLEX parameters to their default values.

```
public void setDeleteMode(IloCplex::DeleteMode mode)
```

This method sets the delete mode in the invoking `IloCplex` object to `mode`.

```
public void setDirection(IloNumVar var, IloCplex::BranchDirection dir)
public void setDirection(IloIntVar var, IloCplex::BranchDirection dir)
```

This method sets the preferred branching direction for variable `var` to `dir`. This setting will cause CPLEX first to explore the branch specified by `dir` after branching on variable `var`.

```
public void setDirections(const IloNumVarArray var, const
IloCplex::BranchDirectionArray dir)
public void setDirections(const IloIntVarArray var, const
IloCplex::BranchDirectionArray dir)
```

This method sets the preferred branching direction for each variable in the array `var` to the corresponding value in the array `dir`. This will cause CPLEX first to explore the branch specified by `dir[i]` after branching on variable `var[i]`.

```
public void setParam(IloCplex::BoolParam parameter, IloBool value)
public void setParam(IloCplex::StringParam parameter, const char * value)
public void setParam(IloCplex::NumParam parameter, IloNum value)
public void setParam(IloCplex::IntParam parameter, IloInt value)
```

This method sets `parameter` to `value` in the invoking algorithm. See the *CPLEX User's Manual* for more detailed information about parameters and for examples of their use.

```
public void setParameterSet(IloCplex::ParameterSet set)
```

Sets the parameter state using a parameter set.

If the method fails, an exception of type `IloException`, or one of its derived classes, is thrown.

Parameters:

set The parameter set.

```
public void setPriorities(const IloNumVarArray var, const IloNumArray pri)
public void setPriorities(const IloIntVarArray var, const IloNumArray pri)
```

This method sets the priority order for all variables in the array `var` to the corresponding value in the array `pri`. During branching, integer variables with higher priorities are given preference over integer variables with lower priorities. Further, variables that have priority assigned to them are given preference over variables that do not. Priorities must be nonnegative integers. By default, the priority of a variable without a user-assigned priority is 0 (zero). The parameter `MIPOrdInd` by default specifies that user-assigned priority orders should be taken into account. When `MIPOrdInd` is reset to its nondefault value 0 (zero), CPLEX ignores user-assigned priorities. To remove user-assigned priority from a variable, see the method `IloCplex::delPriorities`. For more detail about how priorities are applied, see the topic *Issuing Priority Orders* in the *CPLEX User's Manual*.

```
public void setPriority(IloNumVar var, IloNum pri)
public void setPriority(IloIntVar var, IloNum pri)
```

This method sets the priority order for the variable `var` to `pri`. During branching, integer variables with higher priorities are given preference over integer variables with lower priorities. Further, variables that have priority assigned to them are given preference over variables that do not. Priorities must be nonnegative integers. By default, the priority of a variable without a user-assigned priority is 0 (zero). The parameter `MIPOrdInd` by default specifies that user-assigned priority orders should be taken into account. When `MIPOrdInd` is reset to its nondefault value 0 (zero), CPLEX ignores user-assigned priorities. To remove user-assigned priority from a variable, see the method `IloCplex::delPriority`. For more detail about how priorities are applied, see the topic *Issuing Priority Orders* in the *CPLEX User's Manual*.

```
public void setVectors(const IloNumArray x, const IloNumArray dj, const
IloNumVarArray var, const IloNumArray slack, const IloNumArray pi, const
IloRangeArray rng)
public void setVectors(const IloNumArray x, const IloNumArray dj, const
IloIntVarArray var, const IloNumArray slack, const IloNumArray pi, const
IloRangeArray rng)
```

This method allows a user to specify a starting point for the following invocation of the method `IloCplex::solve`.

For all variables in `var`, `x[i]` specifies the starting value for the variable `var[i]`. Similarly, `dj[i]` specifies the starting reduced cost for variable `var[i]`. For all ranged constraints specified in `rng`, `slack[i]` specifies the starting slack value for `rng[i]`. Similarly, `pi[i]` specifies the starting dual value for `rng[i]`.

Zero can be passed for any individual parameter. However, the arrays `x` and `var` must be the same length. Likewise, `pi` and `rng` must be the same length.

In other words, you must provide starting values for either the primal or dual variables `x` and `pi`. If you provide values for `dj`, then you must provide the corresponding values for `x`. If you provide values for `slack`, then you must provide the corresponding values for `pi`.

This information is exploited at the next call to `IloCplex::solve`, to construct a starting point for the algorithm, provided that the `AdvInd` parameter is set to a value greater than or equal to one. In particular, if the extracted model is an LP, and the root algorithm is dual or primal, the information is used to construct a starting basis for the simplex method for the original model, if `AdvInd` is set to 1 (one). If `AdvInd` is set to 2, the information is used to construct a starting basis for the presolved model.

If the extracted model is a MIP, only `x` values can be used. Values may be specified for any subset of the integer and continuous variables in the model, either through a single invocation of `setVectors`, or incrementally through multiple calls.

If this method is called several time with different values for the same variable, then the last value replaces the previous values. If the value specified for a variable is greater than or equal to `CPX_INFBOUND`, then no value is set for that variable.

When optimization commences or resumes, CPLEX will attempt to find a feasible MIP solution that is compatible with the set of specified `x` values. When start values are not provided for all integer variables, CPLEX tries to extend the partial solution to a complete solution by solving a MIP on the unspecified variables. The parameter `SubMIPNodeLim` controls the amount of effort CPLEX expends in trying to solve this secondary MIP. If CPLEX finds a complete feasible solution, that solution becomes the incumbent. If the specified values are infeasible, they are retained for use in a subsequent solution repair heuristic. The amount of effort spent in this heuristic can be controlled by the parameter `RepairTries`.

```
public IloBool solve(IloCplex::Goal goal)
```

This method initializes the goal stack of the root node with `goal` before starting the branch-and-cut search. The search tree will be defined by the execute method of `goal` and its subgoals. See the concept Goals and the nested class `IloCplex::GoalI` for more information.

```
public IloBool solve()
```

This method solves the model currently extracted to the invoking `IloCplex` object. The method returns `IloTrue` if it finds a solution (not necessarily an optimal one).

```
public IloBool solveFixed(IloInt soln=IloCplex::IncumbentId)
```

After the invoking algorithm has solved the extracted MIP model to a feasible (but not necessarily optimal) solution as a MIP, this member function solves the relaxation of the model obtained by fixing all the integer variables of the extracted MIP to the values of a solution. The current solution is used if the `soln` argument is omitted or given the value -1; otherwise, the solution pool member indexed by `soln` is used.

A call to this method causes CPLEX to view the extracted MIP model as a continuous model, so that you can obtain information normally available for a continuous solution but normally unavailable for MIP models. CPLEX

continues to view the model as continuous until a call to another method restores the model as a MIP. For example, a call to `IloCplex::solve` restores the model back to MIP and solves the model immediately, provided the advanced start parameter has not been disabled.

```
public IloInt tuneParam(IloArray< const char * > filename, IloCplex::ParameterSet
fixedset)
public IloInt tuneParam(IloArray< const char * > filename)
public IloInt tuneParam(IloCplex::ParameterSet fixedset)
public IloInt tuneParam()
```

The method `tuneParam` tunes the parameters of an instance of `IloCplex` for improved optimizer performance on the current model, or a set of models if the `filename` argument is used. Tuning is carried out by CPLEX making a number of trial runs with a variety parameter settings. Parameters and associated values which should not be changed by the tuning process are specified in the parameter set `fixedset`.

After `tuneParam` has finished, the `IloCplex` parameters will be set to the tuned and fixed settings which can be queried or written to a file. There will not be a solution to the current model.

The parameter `TuningRepeat` specifies how many problem variations for CPLEX to try while tuning when tuning the current model. Using a number of variations can give more robust results when tuning is applied to a single model.

Note that the tuning evaluation measure is meaningful only when `TuningRepeat` is larger than one or when a set of models is being tuned.

A few of the parameter settings control the tuning process. They are specified in the table below; other parameter settings are ignored.

Parameter	Use
`TiLim`	Limits the total time spent tuning
`TuningTiLim`	Limits the time of each trial run
`TuningMeasure`	Controls the tuning evaluation measure
`TuningRepeat`	Sets the number of repeated problem variations
`TuningDisplay`	Controls the level of the tuning display

All callbacks, except the tuning callback, will be ignored. Tuning will monitor the method `abort` and terminate when an abort has been issued.

Returns:

`IloInt` value specifying the completion status of the tuning. The values returned are from the enumeration `TuningStatus`.

```
public IloCplex::Callback use(IloCplex::Callback cb)
```

This method instructs the invoking `IloCplex` object to use `cb` as a callback. If a callback of the same type as `cb` is already being used by the invoking `IloCplex` object, the previously used callback will be overridden. If the callback object `cb` is already being used by another `IloCplex` object, a copy of `cb` will be used instead. A handle to the callback that is installed in the invoking `IloCplex` object is returned. See `IloCplex::CallbackI` for a discussion of how to implement callbacks.

```
public IloCplex::Aborter use(IloCplex::Aborter abort)
```

This method instructs the invoking `IloCplex` object to use the aborter object `abort` to control termination of its solving and tuning methods. If an aborter is already being used by the invoking `IloCplex` object, the previously used aborter will be overridden. A handle to the aborter that is installed in the invoking `IloCplex` object is returned.

See Also: IloCplex::Aborter

```
public void writeBasis(const char * name) const
```

Writes the current simplex basis to the file specified by `name`.

By convention, the file extension is `.bas`. The BAS file format is documented in the reference manual *CPLEX File Formats*.

```
public void writeConflict(const char * filename) const
```

Writes a conflict file named `filename`.

```
public void writeFilters(const char * name)
```

Writes filters from the invoking model to a file in FLT format. This format is documented in the reference manual *CPLEX File Formats*.

```
public void writeMIPStart(const char * name, IloInt mst=0) const
```

This method has been **deprecated**. Use the method `IloCplex::writeMIPStarts` instead.

```
public void writeMIPStarts(const char * name, IloInt first=0, IloInt num=IloIntMax)
const
```

Writes MIP start information to the file denoted by `name`. The designated MIP starts begin with the one designated by the index `first` and continue to the specified number `num` of members. All MIP starts will be written if the `first` and `num` arguments are omitted. The single MIP start designated by `first` will be written if the `num` argument is omitted.

By convention, the file extension is `.mst`. The MST file format is documented in the reference manual *CPLEX File Formats* as well as the stylesheet `solution.xsl` and schema `solution.xsd` found in the `include` directory of the product.

The parameter `WriteLevel` tells CPLEX the level of detail to write to the file. The enumeration `IloCplex::WriteLevelType` lists values that may be used.

```
public void writeOrder(const char * filename) const
```

Writes a priority order to the file `filename`.

If a priority order has been associated with the CPLEX problem object, or the parameter `MipOrdType` is nonzero, this method writes the priority order into the specified file.

By convention, the file extension is `.ord`. The ORD file format is documented in the reference manual *CPLEX File Formats*.

```
public void writeParam(const char * name) const
```

Writes the parameter name and its current setting into the file specified by `name` for all the CPLEX parameters that are not currently set at their default.

By convention, the file extension is `.prm`. The PRM file format is documented in the reference manual *CPLEX File Formats*.

```
public void writeSolution(const char * name, IloInt soln=IncumbentId) const
```

The method writes a solution for the current problem into the file specified by `name`. The solution written is the current solution if the argument `soln` is omitted; otherwise, it is the solution pool member indexed by `soln`.

By convention, the file extension is `.sol`. The SOL file format is documented in the reference manual *CPLEX File Formats* as well as the stylesheet `solution.xsl` and schema `solution.xsd` found in the include directory of the product.

The parameter `WriteLevel` tells CPLEX the level of detail to write to the file. The enumeration `IloCplex::WriteLevelType` lists values that may be used.

```
public void writeSolutions(const char * name) const
```

Writes a formatted file containing all members of the solution pool for the current problem into the file specified by `name` and does so in SOL format.

By convention, the file extension is `.sol`. The SOL file format is documented in the reference manual *CPLEX File Formats* as well as the stylesheet `solution.xsl` and schema `solution.xsd` found in the include directory of the product.

The parameter `WriteLevel` tells CPLEX the level of detail to write to the file. The enumeration `IloCplex::WriteLevelType` lists values that may be used.

Inner Enumerations

Enumeration Algorithm

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::Algorithm` lists the algorithms available in CPLEX to solve continuous models as controlled by the parameters `IloCplex::RootAlg` and `IloCplex::NodeAlg`.

These values are also returned by `IloCplex::getAlgorithm` to specify the algorithm used to generate the current solution. The values `FeasOpt` and `MIP` are returned by `IloCplex::getAlgorithm` but should **not** be used with `IloCplex::RootAlg` nor with `IloCplex::NodeAlg`.

See Also: IloCplex, IloCplex::getAlgorithm, IloCplex::getSubAlgorithm, IloCplex::IntParam, RootAlg, NodeAlg

Fields:

`NoAlg = CPX_ALG_NONE`	= CPX_ALG_NONE

`AutoAlg = CPX_ALG_AUTOMATIC` = CPX_ALG_AUTOMATIC

`Primal = CPX_ALG_PRIMAL` = CPX_ALG_PRIMAL

`Dual = CPX_ALG_DUAL` = CPX_ALG_DUAL

`Barrier = CPX_ALG_BARRIER` = CPX_ALG_BARRIER

`Sifting = CPX_ALG_SIFTING` = CPX_ALG_SIFTING

`Concurrent = CPX_ALG_CONCURRENT` = CPX_ALG_CONCURRENT

`Network = CPX_ALG_NET` = CPX_ALG_NET

`FeasOpt = CPX_ALG_FEASOPT` = CPX_ALG_FEASOPT

`MIP = CPX_ALG_MIP` = CPX_ALG_MIP

Enumeration BasisStatus

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::BasisStatus` lists values that the status of variables or range constraints may assume in a basis. `NotABasicStatus` is not a valid status for a variable. A basis containing such a status does not constitute a valid basis. The basis status of a ranged constraint corresponds to the basis status of the corresponding slack or artificial variable that `IloCplex` manages for it. `FreeOrSuperbasic` specifies that the variable is nonbasic, but not at a bound.

See Also: IloCplex, IloCplex::BasisStatusArray

Fields:

`NotABasicStatus = -1`

`Basic = 1`

`AtLower = 0`

`AtUpper = 2`

`FreeOrSuperbasic = 3`

Enumeration BoolParam

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::BoolParam` lists the parameters of CPLEX that require Boolean values. Boolean values are also known in certain contexts as binary values or as zero-one (0-1) values. Use these values with the methods that accept Boolean parameters: `IloCplex::getParam` and `IloCplex::setParam`.

See the *CPLEX Parameters Reference Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`PreInd = CPX_PARAM_PREIND` = CPX_PARAM_PREIND

`ReverseInd = CPX_PARAM_REVERSEIND` = deprecated

`XXXInd = CPX_PARAM_XXXIND`	= deprecated
`MIPOrdInd = CPX_PARAM_MIPORDIND`	= CPX_PARAM_MIPORDIND
`MPSLongNum = CPX_PARAM_MPSLONGNUM`	= CPX_PARAM_MPSLONGNUM
`LBHeur = CPX_PARAM_LBHEUR`	= CPX_PARAM_LBHEUR
`PerInd = CPX_PARAM_PERIND`	= CPX_PARAM_PERIND
`PreLinear = CPX_PARAM_PRELINEAR`	= CPX_PARAM_PRELINEAR
`DataCheck = CPX_PARAM_DATACHECK`	= CPX_PARAM_DATACHECK
`QPmakePSDInd = CPX_PARAM_QPMAKEPSDIND`	= CPX_PARAM_QPMAKEPSDIND
`MemoryEmphasis = CPX_PARAM_MEMORYEMPHASIS`	= CPX_PARAM_MEMORYEMPHASIS
`NumericalEmphasis = CPX_PARAM_NUMERICALEMPHASIS`	= CPX_PARAM_NUMERICALEMPHASIS

Enumeration BranchDirection

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::BranchDirection` lists values that can be used for specifying branch directions either with the branch direction parameter `IloCplex::BrDir` or with the methods `IloCplex::setDirection` and `IloCplex::setDirections`. The branch direction specifies which direction to explore first after branching on one variable.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex, IloCplex::BranchDirectionArray

Fields:

`BranchGlobal = CPX_BRANCH_GLOBAL`	= CPX_BRANCH_GLOBAL
`BranchDown = CPX_BRANCH_DOWN`	= CPX_BRANCH_DOWN
`BranchUp = CPX_BRANCH_UP`	= CPX_BRANCH_UP

Enumeration ConflictStatus

Definition file: ilcplex/ilocplexi.h

This enumeration lists the values that tell the status of a constraint or bound with respect to a conflict.

- `ConflictPossibleMember`
- `ConflictMember`

The value `ConflictExcluded` is internal, undocumented, not available to users.

Fields:

`ConflictExcluded = CPX_CONFLICT_EXCLUDED`

`ConflictPossibleMember = CPX_CONFLICT_POSSIBLE_MEMBER`

`ConflictMember = CPX_CONFLICT_MEMBER`

Enumeration CplexStatus

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::CplexStatus` lists values that the status of an `IloCplex` algorithm can assume. The methods `IloCplex::getCplexStatus` and `IloCplex::getCplexSubStatus` access the status values, providing information about what the algorithm learned about the active model in the most recent invocation of the method `solve` or `feasOpt`. The status may also tell why the algorithm terminated.

See the group optim.cplex.solutionstatus in the *Callable Library Reference Manual*, where they are listed in alphabetic order, or the topic *Interpreting Solution Status Codes* in the Overview of the APIs, where they are listed in numeric order, for more information about these values. Also see the *CPLEX User's Manual* for examples of their use.

See also the enumeration `IloAlgorithm::Status` in the *Reference Manual* of the C++ API.

See Also: IloCplex

Fields:

`Unknown = 0`

`Optimal = CPX_STAT_OPTIMAL` = CPX_STAT_OPTIMAL

`Unbounded = CPX_STAT_UNBOUNDED` = CPX_STAT_UNBOUNDED

`Infeasible = CPX_STAT_INFEASIBLE` = CPX_STAT_INFEASIBLE

`InfOrUnbd = CPX_STAT_INForUNBD` = CPX_STAT_INForUNBD

`OptimalInfeas = CPX_STAT_OPTIMAL_INFEAS` = CPX_STAT_OPTIMAL_INFEAS

`NumBest = CPX_STAT_NUM_BEST` = CPX_STAT_NUM_BEST

`FeasibleRelaxedSum = CPX_STAT_FEASIBLE_RELAXED_SUM` = CPX_STAT_FEASIBLE_RELAXED_SUM

`OptimalRelaxedSum = CPX_STAT_OPTIMAL_RELAXED_SUM` = CPX_STAT_OPTIMAL_RELAXED_SUM

`FeasibleRelaxedInf = CPX_STAT_FEASIBLE_RELAXED_INF` = CPX_STAT_FEASIBLE_RELAXED_INF

`OptimalRelaxedInf = CPX_STAT_OPTIMAL_RELAXED_INF` = CPX_STAT_OPTIMAL_RELAXED_INF

`FeasibleRelaxedQuad = CPX_STAT_FEASIBLE_RELAXED_QUAD` = CPX_STAT_FEASIBLE_RELAXED_QUAD

`OptimalRelaxedQuad = CPX_STAT_OPTIMAL_RELAXED_QUAD` = CPX_STAT_OPTIMAL_RELAXED_QUAD

`AbortRelaxed = CPXMIP_ABORT_RELAXED` = CPXMIP_ABORT_RELAXED

`AbortObjLim = CPX_STAT_ABORT_OBJ_LIM` = CPX_STAT_ABORT_OBJ_LIM

`AbortPrimObjLim = CPX_STAT_ABORT_PRIM_OBJ_LIM` = CPX_STAT_ABORT_PRIM_OBJ_LIM

`AbortDualObjLim = CPX_STAT_ABORT_DUAL_OBJ_LIM` = CPX_STAT_ABORT_DUAL_OBJ_LIM

`AbortItLim = CPX_STAT_ABORT_IT_LIM` = CPX_STAT_ABORT_IT_LIM

```
AbortTimeLim = CPX_STAT_ABORT_TIME_LIM      = CPX_STAT_ABORT_TIME_LIM
AbortUser = CPX_STAT_ABORT_USER             = CPX_STAT_ABORT_USER
OptimalFaceUnbounded =                      = CPX_STAT_OPTIMAL_FACE_UNBOUNDED
CPX_STAT_OPTIMAL_FACE_UNBOUNDED
OptimalTol = CPXMIP_OPTIMAL_TOL             = CPXMIP_OPTIMAL_TOL
SolLim = CPXMIP_SOL_LIM                     = CPXMIP_SOL_LIM
PopulateSolLim = CPXMIP_POPULATESOL_LIM     = CPXMIP_POPULATESOL_LIM
NodeLimFeas = CPXMIP_NODE_LIM_FEAS          = CPXMIP_NODE_LIM_FEAS
NodeLimInfeas = CPXMIP_NODE_LIM_INFEAS      = CPXMIP_NODE_LIM_INFEAS
FailFeas = CPXMIP_FAIL_FEAS                 = CPXMIP_FAIL_FEAS
FailInfeas = CPXMIP_FAIL_INFEAS             = CPXMIP_FAIL_INFEAS
MemLimFeas = CPXMIP_MEM_LIM_FEAS            = CPXMIP_MEM_LIM_FEAS
MemLimInfeas = CPXMIP_MEM_LIM_INFEAS        = CPXMIP_MEM_LIM_INFEAS
FailFeasNoTree =                            = CPXMIP_FAIL_FEAS_NO_TREE
CPXMIP_FAIL_FEAS_NO_TREE
FailInfeasNoTree =                          = CPXMIP_FAIL_INFEAS_NO_TREE
CPXMIP_FAIL_INFEAS_NO_TREE
ConflictFeasible =                          = CPX_STAT_CONFLICT_FEASIBLE
CPX_STAT_CONFLICT_FEASIBLE
ConflictMinimal =                           = CPX_STAT_CONFLICT_MINIMAL
CPX_STAT_CONFLICT_MINIMAL
ConflictAbortContradiction =                =
CPX_STAT_CONFLICT_ABORT_CONTRADICTION       CPX_STAT_CONFLICT_ABORT_CONTRADICTION
ConflictAbortTimeLim =                      = CPX_STAT_CONFLICT_ABORT_TIME_LIM
CPX_STAT_CONFLICT_ABORT_TIME_LIM
ConflictAbortItLim =                        = CPX_STAT_CONFLICT_ABORT_IT_LIM
CPX_STAT_CONFLICT_ABORT_IT_LIM
ConflictAbortNodeLim =                      = CPX_STAT_CONFLICT_ABORT_NODE_LIM
CPX_STAT_CONFLICT_ABORT_NODE_LIM
ConflictAbortObjLim =                       = CPX_STAT_CONFLICT_ABORT_OBJ_LIM
CPX_STAT_CONFLICT_ABORT_OBJ_LIM
ConflictAbortMemLim =                       = CPX_STAT_CONFLICT_ABORT_MEM_LIM
CPX_STAT_CONFLICT_ABORT_MEM_LIM
ConflictAbortUser =                         = CPX_STAT_CONFLICT_ABORT_USER
CPX_STAT_CONFLICT_ABORT_USER
Feasible = CPX_STAT_FEASIBLE                = CPX_STAT_FEASIBLE
OptimalPopulated =                          = CPXMIP_OPTIMAL_POPULATED
CPXMIP_OPTIMAL_POPULATED
OptimalPopulatedTol =                       = CPXMIP_OPTIMAL_POPULATED_TOL
CPXMIP_OPTIMAL_POPULATED_TOL
```

Enumeration CutType

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::CutType` lists the values that may be used in querying the number of cuts used in a mixed integer optimization with `getNcuts()`.

Fields:

```
CutCover = CPX_CUT_COVER

CutGubCover = CPX_CUT_GUBCOVER

CutFlowCover = CPX_CUT_FLOWCOVER

CutClique = CPX_CUT_CLIQUE

CutFrac = CPX_CUT_FRAC

CutMCF = CPX_CUT_MCF

CutMir = CPX_CUT_MIR

CutFlowPath = CPX_CUT_FLOWPATH

CutDisj = CPX_CUT_DISJ

CutImplBd = CPX_CUT_IMPLBD

CutZeroHalf = CPX_CUT_ZEROHALF

CutLocalCover = CPX_CUT_LOCALCOVER

CutTighten = CPX_CUT_TIGHTEN

CutObjDisj = CPX_CUT_OBJDISJ

CutUser = CPX_CUT_USER

CutTable = CPX_CUT_TABLE

CutSolnPool = CPX_CUT_SOLNPOOL
```

Enumeration DeleteMode

Definition file: ilcplex/ilocplexi.h

This enumeration lists the possible settings for the delete mode of `IloCplex` as controlled by the method `IloCplex::setDeleteMode` and queried by the method `IloCplex::getDeleteMode`.

- `IloCplex::LeaveBasis`

With the default setting `IloCplex::LeaveBasis`, an existing basis will remain unchanged if variables or constraints are removed from the loaded LP model. This choice generally renders the basis unusable for a restart when CPLEX is solving the modified LP and the advanced indicator (parameter `IloCplex::AdvInd`) is set to `IloTrue`.

- `IloCplex::FixBasis`

In contrast, with delete mode set to `IloCplex::FixBasis`, the invoking object will do basis pivots in order to maintain a valid basis when variables or constraints are removed. This choice makes the delete operation more computation-intensive, but may give a better starting point for reoptimization after modification of the extracted model.

If no basis is present in the invoking object, the setting of the delete mode has no effect.

See Also: IloCplex

Fields:

`LeaveBasis`

`FixBasis`

Enumeration DualPricing

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::DualPricing` lists values that the dual pricing parameter `IloCplex:DPriInd` can assume in `IloCplex` for use with the dual simplex algorithm. Use these values with the method `IloCplex::setParam(IloCplex::DPriInd, value)` when you set the dual pricing indicator.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`DPriIndAuto = CPX_DPRIIND_AUTO`	= CPX_DPRIIND_AUTO
`DPriIndFull = CPX_DPRIIND_FULL`	= CPX_DPRIIND_FULL
`DPriIndSteep = CPX_DPRIIND_STEEP`	= CPX_DPRIIND_STEEP
`DPriIndFullSteep = CPX_DPRIIND_FULLSTEEP`	= CPX_DPRIIND_FULLSTEEP
`DPriIndSteepQStart = CPX_DPRIIND_STEEPQSTART`	= CPX_DPRIIND_STEEPQSTART
`DPriIndDevex = CPX_DPRIIND_DEVEX`	= CPX_DPRIIND_DEVEX

Enumeration IntParam

Definition file: ilcplex/ilocplexi.h

`IloCplex` is the class for the algorithms in CPLEX. The enumeration `IloCplex::IntParam` lists the parameters of CPLEX that require integer values. Use these values with the methods `IloCplex::getParam` and `IloCplex::setParam`.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`AdvInd = CPX_PARAM_ADVIND`	= CPX_PARAM_ADVIND
`RootAlg = CPX_PARAM_LPMETHOD`	= CPX_PARAM_STARTALG, CPX_PARAM_LPMETHOD, CPX_PARAM_QPMETHOD
`NodeAlg = CPX_PARAM_SUBALG`	= CPX_PARAM_SUBALG

```
MIPEmphasis = CPX_PARAM_MIPEMPHASIS          = CPX_PARAM_MIPEMPHASIS
AggFill = CPX_PARAM_AGGFILL                  = CPX_PARAM_AGGFILL
AggInd = CPX_PARAM_AGGIND                    = CPX_PARAM_AGGIND
BasInterval = CPX_PARAM_BASINTERVAL          = CPX_PARAM_BASINTERVAL
ClockType = CPX_PARAM_CLOCKTYPE              = CPX_PARAM_CLOCKTYPE
CraInd = CPX_PARAM_CRAIND                    = CPX_PARAM_CRAIND
DepInd = CPX_PARAM_DEPIND                    = CPX_PARAM_DEPIND
PreDual = CPX_PARAM_PREDUAL                  = CPX_PARAM_PREDUAL
PrePass = CPX_PARAM_PREPASS                  = CPX_PARAM_PREPASS
RelaxPreInd = CPX_PARAM_RELAXPREIND          = CPX_PARAM_RELAXPREIND
RepeatPresolve = CPX_PARAM_REPEATPRESOLVE    = CPX_PARAM_REPEATPRESOLVE
Symmetry = CPX_PARAM_SYMMETRY                = CPX_PARAM_SYMMETRY
DPriInd = CPX_PARAM_DPRIIND                  = CPX_PARAM_DPRIIND
PriceLim = CPX_PARAM_PRICELIM                = CPX_PARAM_PRICELIM
SimDisplay = CPX_PARAM_SIMDISPLAY            = CPX_PARAM_SIMDISPLAY
ItLim = CPX_PARAM_ITLIM                      = CPX_PARAM_ITLIM
NetFind = CPX_PARAM_NETFIND                  = CPX_PARAM_NETFIND
PerLim = CPX_PARAM_PERLIM                    = CPX_PARAM_PERLIM
PPriInd = CPX_PARAM_PPRIIND                  = CPX_PARAM_PPRIIND
ReInv = CPX_PARAM_REINV                      = CPX_PARAM_REINV
ScaInd = CPX_PARAM_SCAIND                    = CPX_PARAM_SCAIND
Threads = CPX_PARAM_THREADS                  = CPX_PARAM_THREADS
ParallelMode = CPX_PARAM_PARALLELMODE        = CPX_PARAM_PARALLELMODE
SingLim = CPX_PARAM_SINGLIM                  = CPX_PARAM_SINGLIM
Reduce = CPX_PARAM_REDUCE                    = CPX_PARAM_REDUCE
NzReadLim = CPX_PARAM_NZREADLIM              = CPX_PARAM_NZREADLIM
ColReadLim = CPX_PARAM_COLREADLIM            = CPX_PARAM_COLREADLIM
RowReadLim = CPX_PARAM_ROWREADLIM            = CPX_PARAM_ROWREADLIM
QPNzReadLim = CPX_PARAM_QPNZREADLIM          = CPX_PARAM_QPNZREADLIM
WriteLevel = CPX_PARAM_WRITELEVEL            = CPX_PARAM_WRITELEVEL
SiftDisplay = CPX_PARAM_SIFTDISPLAY          = CPX_PARAM_SIFTDISPLAY
SiftAlg = CPX_PARAM_SIFTALG                  = CPX_PARAM_SIFTALG
SiftItLim = CPX_PARAM_SIFTITLIM              = CPX_PARAM_SIFTITLIM
BrDir = CPX_PARAM_BRDIR                      = CPX_PARAM_BRDIR
Cliques = CPX_PARAM_CLIQUES                  = CPX_PARAM_CLIQUES
CoeRedInd = CPX_PARAM_COEREDIND              = CPX_PARAM_COEREDIND
```

```
Covers = CPX_PARAM_COVERS                    = CPX_PARAM_COVERS
MIPDisplay = CPX_PARAM_MIPDISPLAY            = CPX_PARAM_MIPDISPLAY
MIPInterval = CPX_PARAM_MIPINTERVAL          = CPX_PARAM_MIPINTERVAL
IntSolLim = CPX_PARAM_INTSOLLIM              = CPX_PARAM_INTSOLLIM
NodeFileInd = CPX_PARAM_NODEFILEIND          = CPX_PARAM_NODEFILEIND
NodeLim = CPX_PARAM_NODELIM                  = CPX_PARAM_NODELIM
NodeSel = CPX_PARAM_NODESEL                  = CPX_PARAM_NODESEL
VarSel = CPX_PARAM_VARSEL                    = CPX_PARAM_VARSEL
BndStrenInd = CPX_PARAM_BNDSTRENIND          = CPX_PARAM_BNDSTRENIND
HeurFreq = CPX_PARAM_HEURFREQ                = CPX_PARAM_HEURFREQ
RINSHeur = CPX_PARAM_RINSHEUR                = CPX_PARAM_RINSHEUR
FPHeur = CPX_PARAM_FPHEUR                    = CPX_PARAM_FPHEUR
RepairTries = CPX_PARAM_REPAIRTRIES          = CPX_PARAM_REPAIRTRIES
SubMIPNodeLim = CPX_PARAM_SUBMIPNODELIM      = CPX_PARAM_SUBMIPNODELIM
MIPOrdType = CPX_PARAM_MIPORDTYPE            = CPX_PARAM_MIPORDTYPE
BBInterval = CPX_PARAM_BBINTERVAL            = CPX_PARAM_BBINTERVAL
FlowCovers = CPX_PARAM_FLOWCOVERS            = CPX_PARAM_FLOWCOVERS
ImplBd = CPX_PARAM_IMPLBD                    = CPX_PARAM_IMPLBD
Probe = CPX_PARAM_PROBE                      = CPX_PARAM_PROBE
GUBCovers = CPX_PARAM_GUBCOVERS              = CPX_PARAM_GUBCOVERS
StrongCandLim = CPX_PARAM_STRONGCANDLIM      = CPX_PARAM_STRONGCANDLIM
StrongItLim = CPX_PARAM_STRONGITLIM          = CPX_PARAM_STRONGITLIM
FracCand = CPX_PARAM_FRACCAND                = CPX_PARAM_FRACCAND
FracCuts = CPX_PARAM_FRACCUTS                = CPX_PARAM_FRACCUTS
FracPass = CPX_PARAM_FRACPASS                = CPX_PARAM_FRACPASS
PreslvNd = CPX_PARAM_PRESLVND                = CPX_PARAM_PRESLVND
FlowPaths = CPX_PARAM_FLOWPATHS              = CPX_PARAM_FLOWPATHS
MIRCuts = CPX_PARAM_MIRCUTS                  = CPX_PARAM_MIRCUTS
DisjCuts = CPX_PARAM_DISJCUTS                = CPX_PARAM_DISJCUTS
ZeroHalfCuts = CPX_PARAM_ZEROHALFCUTS        = CPX_PARAM_ZEROHALFCUTS
MCFCuts = CPX_PARAM_MCFCUTS                  = CPX_PARAM_MCFCUTS
AggCutLim = CPX_PARAM_AGGCUTLIM              = CPX_PARAM_AGGCUTLIM
CutPass = CPX_PARAM_CUTPASS                  = CPX_PARAM_CUTPASS
EachCutLim = CPX_PARAM_EACHCUTLIM            = CPX_PARAM_EACHCUTLIM
DiveType = CPX_PARAM_DIVETYPE                = CPX_PARAM_DIVETYPE
MIPSearch = CPX_PARAM_MIPSEARCH              = CPX_PARAM_MIPSEARCH
```

`MIQCPStrat = CPX_PARAM_MIQCPSTRAT`	= CPX_PARAM_MIQCPSTRAT
`SolnPoolCapacity = CPX_PARAM_SOLNPOOLCAPACITY`	= CPX_PARAM_SOLNPOOLCAPACITY
`SolnPoolReplace = CPX_PARAM_SOLNPOOLREPLACE`	= CPX_PARAM_SOLNPOOLREPLACE
`SolnPoolIntensity = CPX_PARAM_SOLNPOOLINTENSITY`	= CPX_PARAM_SOLNPOOLINTENSITY
`PopulateLim = CPX_PARAM_POPULATELIM`	= CPX_PARAM_POPULATELIM
`BarAlg = CPX_PARAM_BARALG`	= CPX_PARAM_BARALG
`BarColNz = CPX_PARAM_BARCOLNZ`	= CPX_PARAM_BARCOLNZ
`BarDisplay = CPX_PARAM_BARDISPLAY`	= CPX_PARAM_BARDISPLAY
`BarItLim = CPX_PARAM_BARITLIM`	= CPX_PARAM_BARITLIM
`BarMaxCor = CPX_PARAM_BARMAXCOR`	= CPX_PARAM_BARMAXCOR
`BarOrder = CPX_PARAM_BARORDER`	= CPX_PARAM_BARORDER
`BarCrossAlg = CPX_PARAM_BARCROSSALG`	= CPX_PARAM_BARCROSSALG
`BarStartAlg = CPX_PARAM_BARSTARTALG`	= CPX_PARAM_BARSTARTALG
`NetItLim = CPX_PARAM_NETITLIM`	= CPX_PARAM_NETITLIM
`NetPPriInd = CPX_PARAM_NETPPRIIND`	= CPX_PARAM_NETPPRIIND
`NetDisplay = CPX_PARAM_NETDISPLAY`	= CPX_PARAM_NETDISPLAY
`ConflictDisplay = CPX_PARAM_CONFLICTDISPLAY`	= CPX_PARAM_CONFLICTDISPLAY
`FeasOptMode = CPX_PARAM_FEASOPTMODE`	= CPX_PARAM_FEASOPTMODE
`TuningMeasure = CPX_PARAM_TUNINGMEASURE`	= CPX_PARAM_TUNINGMEASURE
`TuningRepeat = CPX_PARAM_TUNINGREPEAT`	= CPX_PARAM_TUNINGREPEAT
`TuningDisplay = CPX_PARAM_TUNINGDISPLAY`	= CPX_PARAM_TUNINGDISPLAY
`PolishAfterNode = CPX_PARAM_POLISHAFTERNODE`	= CPX_PARAM_POLISHAFTERNODE
`PolishAfterIntSol = CPX_PARAM_POLISHAFTERINTSOL`	= CPX_PARAM_POLISHAFTERINTSOL

Enumeration MIPEmphasisType

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::MIPEmphasisType` lists the values that the MIP emphasis parameter `IloCplex::MIPEmphasis` can assume in an instance of `IloCplex` for use when it is solving MIP problems. Use these values with the method `IloCplex::setParam(IloCplex::MIPEmphasis, value)` when you set MIP emphasis.

With the default setting of `IloCplex::MIPEmphasisBalance`, `IloCplex` tries to compute the branch-and-cut algorithm in such a way as to find a proven optimal solution quickly. For a discussion about various settings, refer to the *CPLEX User's Manual*.

See the *CPLEX Parameters Reference Manual* and the information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`MIPEmphasisBalanced = CPX_MIPEMPHASIS_BALANCED`	= CPX_MIPEMPHASIS_BALANCED
`MIPEmphasisOptimality = CPX_MIPEMPHASIS_OPTIMALITY`	= CPX_MIPEMPHASIS_OPTIMALITY
`MIPEmphasisFeasibility = CPX_MIPEMPHASIS_FEASIBILITY`	= CPX_MIPEMPHASIS_FEASIBILITY
`MIPEmphasisBestBound = CPX_MIPEMPHASIS_BESTBOUND`	= CPX_MIPEMPHASIS_BESTBOUND
`MIPEmphasisHiddenFeas = CPX_MIPEMPHASIS_HIDDENFEAS`	= CPX_MIPEMPHASIS_HIDDENFEAS

Enumeration MIPsearch

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::MIPsearch` lists values that the dynamic search parameter `IloCplex::MIPSearch` can assume in `IloCplex`. Use these values with the method IloCplex::setParam `(IloCplex::MIPSearch, value)`.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about this parameter. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`AutoSearch = CPX_MIPSEARCH_AUTO`	= CPX_MIPSEARCH_AUTO
`Traditional = CPX_MIPSEARCH_TRADITIONAL`	= CPX_MIPSEARCH_TRADITIONAL
`Dynamic = CPX_MIPSEARCH_DYNAMIC`	= CPX_MIPSEARCH_DYNAMIC

Enumeration MIPStartEffort

Definition file: ilcplex/ilocplexi.h

This enumeration defines the possible levels of effort for CPLEX to expend when CPLEX evaluates a MIP start.

- Level 0 (zero) MIPStartAuto: Automatic, let CPLEX decide.
- Level 1 (one) MIPStartCheckFeas: CPLEX checks the feasibility of the MIP start.
- Level 2 MIPStartSolveFixed: CPLEX solves the fixed problem specified by the MIP start.
- Level 3 MIPStartSolveMIP: CPLEX solves a subMIP.
- Level 4 MIPStartRepair:CPLEX attempts to repair the MIP start if it is infeasible, according to the parameter that sets the frequency to try to repair infeasible MIP start, `CPX_PARAM_REPAIRTRIES`.

Fields:

`MIPStartAuto = CPX_MIPSTART_AUTO`

`MIPStartCheckFeas = CPX_MIPSTART_CHECKFEAS`

`MIPStartSolveFixed = CPX_MIPSTART_SOLVEFIXED`

`MIPStartSolveMIP = CPX_MIPSTART_SOLVEMIP`

`MIPStartRepair = CPX_MIPSTART_REPAIR`

Enumeration NodeSelect

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::NodeSelect` lists values that the parameter `IloCplex::NodeSel` can assume in `IloCplex`. Use these values with the method `IloCplex::setParam(IloCplex::NodeSel, value)`.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`DFS = CPX_NODESEL_DFS`	= CPX_NODESEL_DFS
`BestBound = CPX_NODESEL_BESTBOUND`	= CPX_NODESEL_BESTBOUND
`BestEst = CPX_NODESEL_BESTEST`	= CPX_NODESEL_BESTEST
`BestEstAlt = CPX_NODESEL_BESTEST_ALT`	= CPX_NODESEL_BESTEST_ALT

Enumeration NumParam

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::NumParam` lists the parameters of CPLEX that require numeric values. Use these values with the member functions: `IloCplex::getParam` and `IloCplex::setParam`.

See the *CPLEX Parameters Reference Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for more examples of their use.

See Also: IloCplex

Fields:

`EpMrk = CPX_PARAM_EPMRK`	= CPX_PARAM_EPMRK
`EpOpt = CPX_PARAM_EPOPT`	= CPX_PARAM_EPOPT
`EpPer = CPX_PARAM_EPPER`	= CPX_PARAM_EPPER
`EpRHS = CPX_PARAM_EPRHS`	= CPX_PARAM_EPRHS
`NetEpOpt = CPX_PARAM_NETEPOPT`	= CPX_PARAM_NETEPOPT
`NetEpRHS = CPX_PARAM_NETEPRHS`	= CPX_PARAM_NETEPRHS
`TiLim = CPX_PARAM_TILIM`	= CPX_PARAM_TILIM
`TuningTiLim = CPX_PARAM_TUNINGTILIM`	= CPX_PARAM_TUNINGTILIM
`BtTol = CPX_PARAM_BTTOL`	= CPX_PARAM_BTTOL
`CutLo = CPX_PARAM_CUTLO`	= CPX_PARAM_CUTLO
`CutUp = CPX_PARAM_CUTUP`	= CPX_PARAM_CUTUP
`EpGap = CPX_PARAM_EPGAP`	= CPX_PARAM_EPGAP
`EpInt = CPX_PARAM_EPINT`	= CPX_PARAM_EPINT
`EpAGap = CPX_PARAM_EPAGAP`	= CPX_PARAM_EPAGAP

```
EpRelax = CPX_PARAM_EPRELAX                          = CPX_PARAM_EPRELAX
ObjDif = CPX_PARAM_OBJDIF                            = CPX_PARAM_OBJDIF
ObjLLim = CPX_PARAM_OBJLLIM                          = CPX_PARAM_OBJLLIM
ObjULim = CPX_PARAM_OBJULIM                          = CPX_PARAM_OBJULIM
PolishTime = CPX_PARAM_POLISHTIME                    = CPX_PARAM_POLISHTIME
PolishAfterEpAGap = CPX_PARAM_POLISHAFTEREPAGAP      = CPX_PARAM_POLISHAFTEREPAGAP
PolishAfterEpGap = CPX_PARAM_POLISHAFTEREPGAP        = CPX_PARAM_POLISHAFTEREPGAP
PolishAfterTime = CPX_PARAM_POLISHAFTERTIME          = CPX_PARAM_POLISHAFTERTIME
ProbeTime = CPX_PARAM_PROBETIME                      = CPX_PARAM_PROBETIME
RelObjDif = CPX_PARAM_RELOBJDIF                      = CPX_PARAM_RELOBJDIF
CutsFactor = CPX_PARAM_CUTSFACTOR                    = CPX_PARAM_CUTSFACTOR
TreLim = CPX_PARAM_TRELIM                            = CPX_PARAM_TRELIM
SolnPoolGap = CPX_PARAM_SOLNPOOLGAP                  = CPX_PARAM_SOLNPOOLGAP
SolnPoolAGap = CPX_PARAM_SOLNPOOLAGAP                = CPX_PARAM_SOLNPOOLAGAP
WorkMem = CPX_PARAM_WORKMEM                          = CPX_PARAM_WORKMEM
BarEpComp = CPX_PARAM_BAREPCOMP                      = CPX_PARAM_BAREPCOMP
BarQCPEpComp = CPX_PARAM_BARQCPEPCOMP                = CPX_PARAM_BARQCPEPCOMP
BarGrowth = CPX_PARAM_BARGROWTH                      = CPX_PARAM_BARGROWTH
BarObjRng = CPX_PARAM_BAROBJRNG                      = CPX_PARAM_BAROBJRNG
EpLin = CPX_PARAM_EPLIN                              = CPX_PARAM_EPLIN
```

Enumeration Parallel_Mode

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::ParallelMode` lists values that the parallel mode parameter `IloCplex::ParallelMode` can assume in `IloCplex` for use on multiprocessor or multithread platforms, if your application is licensed for parallel optimization. Use these values with the method IloCplex::setParam `(IloCplex::ParallelMode, value)`.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about this parameter. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

```
Opportunistic = CPX_PARALLEL_OPPORTUNISTIC       = CPX_PARALLEL_OPPORTUNISTIC
AutoParallel = CPX_PARALLEL_AUTO                 = CPX_PARALLEL_AUTO
Deterministic = CPX_PARALLEL_DETERMINISTIC       = CPX_PARALLEL_DETERMINISTIC
```

Enumeration PrimalPricing

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::PrimalPricing` lists values that the primal pricing parameter `IloCplex::PPriInd` can assume in `IloCplex` for use with the primal simplex algorithm. Use these values with the method IloCplex::setParam `(IloCplex::PPriInd, value)` when setting the primal pricing indicator.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`PPriIndPartial = CPX_PPRIIND_PARTIAL`	= CPX_PPRIIND_PARTIAL
`PPriIndAuto = CPX_PPRIIND_AUTO`	= CPX_PPRIIND_AUTO
`PPriIndDevex = CPX_PPRIIND_DEVEX`	= CPX_PPRIIND_DEVEX
`PPriIndSteep = CPX_PPRIIND_STEEP`	= CPX_PPRIIND_STEEP
`PPriIndSteepQStart = CPX_PPRIIND_STEEPQSTART`	= CPX_PPRIIND_STEEPQSTART
`PPriIndFull = CPX_PPRIIND_FULL`	= CPX_PPRIIND_FULL

Enumeration Quality

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::Quality` lists types of quality measures that can be queried for a solution with the method `IloCplex::getQuality`.

See the group optim.cplex.solutionquality in the *Callable Library Reference Manual* for more information about these values. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`MaxPrimalInfeas = CPX_MAX_PRIMAL_INFEAS`	= CPX_MAX_PRIMAL_INFEAS
`MaxScaledPrimalInfeas = CPX_MAX_SCALED_PRIMAL_INFEAS`	= CPX_MAX_SCALED_PRIMAL_INFEAS
`SumPrimalInfeas = CPX_SUM_PRIMAL_INFEAS`	= CPX_SUM_PRIMAL_INFEAS
`SumScaledPrimalInfeas = CPX_SUM_SCALED_PRIMAL_INFEAS`	= CPX_SUM_SCALED_PRIMAL_INFEAS
`MaxDualInfeas = CPX_MAX_DUAL_INFEAS`	= CPX_MAX_DUAL_INFEAS
`MaxScaledDualInfeas = CPX_MAX_SCALED_DUAL_INFEAS`	= CPX_MAX_SCALED_DUAL_INFEAS
`SumDualInfeas = CPX_SUM_DUAL_INFEAS`	= CPX_SUM_DUAL_INFEAS
`SumScaledDualInfeas = CPX_SUM_SCALED_DUAL_INFEAS`	= CPX_SUM_SCALED_DUAL_INFEAS
`MaxIntInfeas = CPX_MAX_INT_INFEAS`	= CPX_MAX_INT_INFEAS

```
SumIntInfeas = CPX_SUM_INT_INFEAS                = CPX_SUM_INT_INFEAS
MaxPrimalResidual = CPX_MAX_PRIMAL_RESIDUAL      = CPX_MAX_PRIMAL_RESIDUAL
MaxScaledPrimalResidual =                        = CPX_MAX_SCALED_PRIMAL_RESIDUAL
CPX_MAX_SCALED_PRIMAL_RESIDUAL
SumPrimalResidual = CPX_SUM_PRIMAL_RESIDUAL      = CPX_SUM_PRIMAL_RESIDUAL
SumScaledPrimalResidual =                        = CPX_SUM_SCALED_PRIMAL_RESIDUAL
CPX_SUM_SCALED_PRIMAL_RESIDUAL
MaxDualResidual = CPX_MAX_DUAL_RESIDUAL          = CPX_MAX_DUAL_RESIDUAL
MaxScaledDualResidual =                          = CPX_MAX_SCALED_DUAL_RESIDUAL
CPX_MAX_SCALED_DUAL_RESIDUAL
SumDualResidual = CPX_SUM_DUAL_RESIDUAL          = CPX_SUM_DUAL_RESIDUAL
SumScaledDualResidual =                          = CPX_SUM_SCALED_DUAL_RESIDUAL
CPX_SUM_SCALED_DUAL_RESIDUAL
MaxCompSlack = CPX_MAX_COMP_SLACK                = CPX_MAX_COMP_SLACK
SumCompSlack = CPX_SUM_COMP_SLACK                = CPX_SUM_COMP_SLACK
MaxX = CPX_MAX_X                                 = CPX_MAX_X
MaxScaledX = CPX_MAX_SCALED_X                    = CPX_MAX_SCALED_X
MaxPi = CPX_MAX_PI                               = CPX_MAX_PI
MaxScaledPi = CPX_MAX_SCALED_PI                  = CPX_MAX_SCALED_PI
MaxSlack = CPX_MAX_SLACK                         = CPX_MAX_SLACK
MaxScaledSlack = CPX_MAX_SCALED_SLACK            = CPX_MAX_SCALED_SLACK
MaxRedCost = CPX_MAX_RED_COST                    = CPX_MAX_RED_COST
MaxScaledRedCost = CPX_MAX_SCALED_RED_COST       = CPX_MAX_SCALED_RED_COST
SumX = CPX_SUM_X                                 = CPX_SUM_X
SumScaledX = CPX_SUM_SCALED_X                    = CPX_SUM_SCALED_X
SumPi = CPX_SUM_PI                               = CPX_SUM_PI
SumScaledPi = CPX_SUM_SCALED_PI                  = CPX_SUM_SCALED_PI
SumSlack = CPX_SUM_SLACK                         = CPX_SUM_SLACK
SumScaledSlack = CPX_SUM_SCALED_SLACK            = CPX_SUM_SCALED_SLACK
SumRedCost = CPX_SUM_RED_COST                    = CPX_SUM_RED_COST
SumScaledRedCost = CPX_SUM_SCALED_RED_COST       = CPX_SUM_SCALED_RED_COST
Kappa = CPX_KAPPA                                = CPX_KAPPA
ObjGap = CPX_OBJ_GAP                             = CPX_OBJ_GAP
DualObj = CPX_DUAL_OBJ                           = CPX_DUAL_OBJ
PrimalObj = CPX_PRIMAL_OBJ                       = CPX_PRIMAL_OBJ
ExactKappa = CPX_EXACT_KAPPA                     = CPX_EXACT_KAPPA
```

Enumeration Relaxation

Definition file: ilcplex/ilocplexi.h

The enumeration `Relaxation` lists the values that can be taken by the parameter `FeasOptMode`. This parameter controls several aspects of how the method `feasOpt` performs its relaxation.

The method `feasOpt` works in two phases. In its first phase, it attempts to find a minimum-penalty relaxation of a given infeasible model. If you want `feasOpt` to stop after this first phase, choose a value with `Min` in its symbolic name. If you want `feasOpt` to continue beyond its first phase to find a solution that is optimal with respect to the original objective function, subject to the constraint that the penalty of the relaxation must not exceed the value found in the first phase, then choose a value with `Opt` in its symbolic name.

In both phases, the suffixes `Sum`, `Inf`, and `Quad` specify the relaxation metric:

- `Sum` tells `feasOpt` to mimimize the weighted sum of the required relaxations of bounds and constraints according to the formula penalty = sum (penalty_i times relaxation_amount_i)
- `Inf` tells `feasOpt` to minimize the weighted number of bounds and constraints that are relaxed according to the formula penalty = sum (penalty_i times relaxation_indicator_i)
- `Quad` tells `feasOpt` to mimimize the weighted sum of the squares of required relaxations of bounds and constraints according to the formula penalty = sum (penalty_i times relaxation_amount_i times relaxation_amount_i)

Weights are determined by the preference values you provide as input to the method `feasOpt`.

When `IloAnd` is used to group constraints as input to `feasOpt`, the relaxation penalty is computed on groups instead of on individual constraints. For example, all constraints in a group can be relaxed for a total penalty of one unit under the various `Inf` metrics.

Fields:

`MinSum = CPX_FEASOPT_MIN_SUM`	= CPX_FEASOPT_MIN_SUM
`OptSum = CPX_FEASOPT_OPT_SUM`	= CPX_FEASOPT_OPT_SUM
`MinInf = CPX_FEASOPT_MIN_INF`	= CPX_FEASOPT_MIN_INF
`OptInf = CPX_FEASOPT_OPT_INF`	= CPX_FEASOPT_OPT_INF
`MinQuad = CPX_FEASOPT_MIN_QUAD`	= CPX_FEASOPT_MIN_QUAD
`OptQuad = CPX_FEASOPT_OPT_QUAD`	= CPX_FEASOPT_OPT_QUAD

Enumeration StringParam

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::StringParam` lists the parameters of CPLEX that require a character string as a value. Use these values with the methods `IloCplex::getParam` and `IloCplex::setParam`.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`WorkDir = CPX_PARAM_WORKDIR`	= CPX_PARAM_WORKDIR

Enumeration TuningStatus

Definition file: ilcplex/ilocplexi.h

This enumeration lists the values that are returned by `tuneParam`.

- `TuningComplete`
- `TuningAbort`
- `TuningTimeLim`

The value `ConflictExcluded` is internal, undocumented, not available to users.

Fields:

`TuningComplete = 0`

`TuningAbort = CPX_TUNE_ABORT`

`TuningTimeLim = CPX_TUNE_TILIM`

Enumeration VariableSelect

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::VariableSelect` lists values that the parameter `IloCplex::VarSel` can assume in `IloCplex`. Use these values with the method `IloCplex::setParam(IloCplex::VarSel, value)`.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`MinInfeas = CPX_VARSEL_MININFEAS` = CPX_VARSEL_MININFEAS

`DefaultVarSel = CPX_VARSEL_DEFAULT` = CPX_VARSEL_DEFAULT

`MaxInfeas = CPX_VARSEL_MAXINFEAS` = CPX_VARSEL_MAXINFEAS

`Pseudo = CPX_VARSEL_PSEUDO` = CPX_VARSEL_PSEUDO

`Strong = CPX_VARSEL_STRONG` = CPX_VARSEL_STRONG

`PseudoReduced = CPX_VARSEL_PSEUDOREDUCED` = CPX_VARSEL_PSEUDOREDUCED

Enumeration WriteLevelType

Definition file: ilcplex/ilocplexi.h

Values of this enumeration specify how much detail CPLEX includes when it writes MIP starts and solutions to a formatted file.

Fields:

`Auto = CPX_WRITELEVEL_AUTO`

`AllVars = CPX_WRITELEVEL_ALLVARS`

`DiscreteVars = CPX_WRITELEVEL_DISCRETEVARS`

`NonzeroVars = CPX_WRITELEVEL_NONZEROVARS`

`NonZeroDiscreteVars = CPX_WRITELEVEL_NONZERODISCRETEVARS`

Inner Typdefs

Typedef BasisStatusArray

Definition file: ilcplex/ilocplexi.h

```
IloArray< BasisStatus > BasisStatusArray
```

This type defines an array-type for `IloCplex::BasisStatus`. The fully qualified name of a basis status array is `IloCplex::BasisStatusArray`.

See Also: IloCplex, IloCplex::BasisStatus

Typedef BranchDirectionArray

Definition file: ilcplex/ilocplexi.h

```
IloArray< BranchDirection > BranchDirectionArray
```

This type defines an array-type for `IloCplex::BranchDirection`. The fully qualified name of a branch direction array is `IloCplex::BranchDirectionArray`.

See Also: IloCplex, IloCplex::BranchDirection

Typedef ConflictStatusArray

Definition file: ilcplex/ilocplexi.h

```
IloArray< ConflictStatus > ConflictStatusArray
```

This type defines an array-type for `IloCplex::ConflictStatus`.

See Also: IloCplex, IloCplex::ConflictStatus

Typedef Status

Definition file: ilcplex/ilocplexi.h

```
CplexStatus Status
```

An enumeration for the class `IloAlgorithm`.
`IloAlgorithm` is the base class of algorithms in Concert Technology, and `IloAlgorithm::Status` is an enumeration limited in scope to the class `IloAlgorithm`. The member function `IloAlgorithm::getStatus` returns a status showing information about the current model and the solution.

`Unknown` specifies that the algorithm has no information about the solution of the model.

`Feasible` specifies that the algorithm found a feasible solution (that is, an assignment of values to variables that satisfies the constraints of the model, though it may not necessarily be optimal). The member functions `IloAlgorithm::getValue` access this feasible solution.

`Optimal` specifies that the algorithm found an optimal solution (that is, an assignment of values to variables that satisfies all the constraints of the model and that is proved optimal with respect to the objective of the model). The member functions `IloAlgorithm::getValue` access this optimal solution.

`Infeasible` specifies that the algorithm proved the model infeasible; that is, it is not possible to find an assignment of values to variables satisfying all the constraints in the model.

`Unbounded` specifies that the algorithm proved the model unbounded.

`InfeasibleOrUnbounded` specifies that the model is infeasible or unbounded.

`Error` specifies that an error occurred and, on platforms that support exceptions, that an exception has been thrown.

See Also: IloAlgorithm, operator<<

Class IloCsvLine

Definition file: ilconcert/ilocsvreader.h

Represents a line in a csv file.
An instance of `IloCsvLine` represents a single line in a file of comma-separated values (csv file).

Constructor and Destructor Summary	
public	IloCsvLine()
public	IloCsvLine(IloCsvLineI * impl)
public	IloCsvLine(const IloCsvLine & csvLine)

Method Summary	
public void	copy(const IloCsvLine)
public IloBool	emptyFieldByHeader(const char * name) const
public IloBool	emptyFieldByPosition(IloInt i) const
public void	end()
public IloNum	getFloatByHeader(const char * name) const
public IloNum	getFloatByHeaderOrDefaultValue(const char * name, IloNum defaultValue) const
public IloNum	getFloatByPosition(IloInt i) const
public IloNum	getFloatByPositionOrDefaultValue(IloInt i, IloNum defaultValue) const
public IloCsvLineI *	getImpl() const
public IloInt	getIntByHeader(const char * name) const
public IloInt	getIntByHeaderOrDefaultValue(const char * name, IloInt defaultValue) const
public IloInt	getIntByPosition(IloInt i) const
public IloInt	getIntByPositionOrDefaultValue(IloInt i, IloInt defaultValue) const
public IloInt	getLineNumber() const
public IloInt	getNumberOfFields() const
public char *	getStringByHeader(const char * name) const
public char *	getStringByHeaderOrDefaultValue(const char * name, const char * defaultValue) const
public char *	getStringByPosition(IloInt i) const
public char *	getStringByPositionOrDefaultValue(IloInt i, const char * defaultValue) const
public void	operator=(const IloCsvLine & csvLine)
public IloBool	printValueOfKeys() const

Constructors and Destructors

`public` **`IloCsvLine`**`()`

This constructor creates a csv line object whose handle pointer is null. This object must be assigned before it can be used.

`public` **`IloCsvLine`**`(IloCsvLineI * impl)`

This constructor creates a handle object (an instance of `IloCsvLine`) from a pointer to an implementation object (an instance of the class `IloCsvLineI`).

`public` **`IloCsvLine`**`(const IloCsvLine & csvLine)`

This copy constructor creates a handle from a reference to a csv line object. The csv line object and `csvLine` both point to the same implementation object.

Methods

`public void` **`copy`**`(const IloCsvLine)`

This member function returns the real number of the invoking csv line in the data file.

`public IloBool` **`emptyFieldByHeader`**`(const char * name) const`

This member function returns `IloTrue` if the field denoted by the string `name` in the invoking csv line is empty. Otherwise, it returns `IloFalse`

`public IloBool` **`emptyFieldByPosition`**`(IloInt i) const`

This member function returns `IloTrue` if the field denoted by `i` in the invoking csv line is empty. Otherwise, it returns `IloFalse`

`public void` **`end`**`()`

This member function deallocates the memory used by the csv line. If you no longer need a csv line, you can call this member function to reduce memory consumption.

`public IloNum` **`getFloatByHeader`**`(const char * name) const`

This member function returns the float contained in the field `name` in the invoking csv line.

If you have a loop in which you are getting a string, integer, or float by header on several lines with the same header name, it is better for performance to get the position of the header named `name` using the member function `IloCsvReader::getPosition(name)` than using `IloCsvLine::getFloatByPosition` (position of name in the header line).

```
public IloNum getFloatByHeaderOrDefaultValue(const char * name, IloNum
defaultValue) const
```

This member function returns the float contained in the field `name` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

```
public IloNum getFloatByPosition(IloInt i) const
```

This member function returns the float contained in the field `i` in the invoking csv line.

```
public IloNum getFloatByPositionOrDefaultValue(IloInt i, IloNum defaultValue) const
```

This member function returns the float contained in the field `i` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

```
public IloCsvLineI * getImpl() const
```

This member function returns a pointer to the implementation object corresponding to the invoking csv line.

```
public IloInt getIntByHeader(const char * name) const
```

This member function returns the integer contained in the field `name` in the invoking csv line.

If you have a loop in which you are getting a string, integer, or float by header on several lines with the same header name, it is better for performance to get the position of the header named `name` using the member function `IloCsvReader::getPosition(name)` than using `IloCsvLine::getIntByPosition` (position of name in the header line).

```
public IloInt getIntByHeaderOrDefaultValue(const char * name, IloInt defaultValue)
const
```

This member function returns the integer contained in the field `name` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

```
public IloInt getIntByPosition(IloInt i) const
```

This member function returns the integer contained in the field `i` in the invoking csv line.

```
public IloInt getIntByPositionOrDefaultValue(IloInt i, IloInt defaultValue) const
```

This member function returns the integer contained in the field `i` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

```
public IloInt getLineNumber() const
```

This member function returns the real number of the invoking csv line in the data file.

```
public IloInt getNumberOfFields() const
```

This member function returns the number of fields in the line.

```
public char * getStringByHeader(const char * name) const
```

This member function returns a reference to the string contained in the field `name` in the invoking csv line.

If you have a loop in which you are getting a string, integer, or float by header on several lines with the same header name, it is better for performance to get the position of the header named `name` using the member function `IloCsvReader::getPosition(name)` than using `IloCsvLine::getStringByPosition` (position of name in the header line).

```
public char * getStringByHeaderOrDefaultValue(const char * name, const char *
defaultValue) const
```

This member function returns the string contained in the field `name` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

```
public char * getStringByPosition(IloInt i) const
```

This member function returns a reference to the string contained in the field number `i` in the invoking csv line.

```
public char * getStringByPositionOrDefaultValue(IloInt i, const char *
defaultValue) const
```

This member function returns the string contained in the field `i` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

```
public void operator=(const IloCsvLine & csvLine)
```

This operator assigns an address to the handle pointer of the invoking csv line. This address is the location of the implementation object of the argument `csvLine`.

After execution of this operator, the invoking csv line and `csvLine` both point to the same implementation object.

```
public IloBool printValueOfKeys() const
```

This member function prints the values of the keys fields in this line.

Class IloCsvReader

Definition file: ilconcert/ilocsvreader.h

Reads a formatted csv file.
An instance of `IloCsvReader` reads a file of comma-separated values of a specified format. The csv file can be a multitable or a single table file. Empty lines and commented lines are allowed everywhere in the file.

Format of multitable files

The first column of the table must contain the name of the table.

Each table can begin with a line containing column headers, the first field of this line must have this format: `tableName|NAMES`

The keys can be specified in the data file by adding a line at the beginning of the table. This line is formatted as follows:

- the first field is `tableName|KEYS`
- the other fields have the value 1 if the corresponding column is a key for the table; if not they have the value 0.

If this line doesn't exist, all columns form a key. If you need to get a line having a specific value for a field, you must add the key line in which you specify that this field is a key for the table.

Any line containing '|' in its first field is ignored by the reader.

A table can be split in several parts in the file (for example, you have a part of table TA, then table TB, then the end of table TA).

Example

```
NODES|NAMES,node_type,node_name,xcoord,ycoord
NODES|KEYS,1,1,0,0
NODES,1,node1,0,1
NODES,1,node2,0,2
NODES,2,node1,0,4
```

Format of single table files

The line containing the column headers, if it exists, must have a first field of the following format: `Field|NAMES`.

Table keys can be specified by adding a line at the beginning of the table. This line must have a first field with this format: `tableName|KEYS`. If this line doesn't exist, all columns form a key.

Example

```
Field|NAMES,nodeName,xCoord,yCoord
Field|KEYS,1,0,0
node1,0,1
node2,0,2
```

Constructor and Destructor Summary	
public	IloCsvReader()
public	IloCsvReader(IloCsvReaderI * impl)

public	IloCsvReader(const IloCsvReader & csv)
public	IloCsvReader(IloEnv env, const char * problem, IloBool multiTable=IloFalse, IloBool allowTableSplitting=IloFalse, const char * separator=",;\t", const char decimalp='.', const char quote='\"', const char comment='#')

Method Summary	
public void	end()
public IloNum	getCsvFormat()
public IloCsvLine	getCurrentLine() const
public IloEnv	getEnv() const
public IloNum	getFileVersion()
public IloCsvReaderI *	getImpl() const
public IloCsvLine	getLineByKey(IloInt numberOfKeys, const char *, ...)
public IloCsvLine	getLineByNumber(IloInt i)
public IloInt	getNumberOfColumns()
public IloInt	getNumberOfItems()
public IloInt	getNumberOfKeys() const
public IloInt	getNumberOfTables()
public IloInt	getPosition(const char * headingName) const
public IloCsvTableReader	getReaderForUniqueTableFile() const
public const char *	getRequiredBy()
public IloCsvTableReader	getTable()
public IloCsvTableReader	getTableByName(const char * name)
public IloCsvTableReader	getTableByNumber(IloInt i)
public IloBool	isHeadingExists(const char * headingName) const
public void	operator=(const IloCsvReader & csv)
public IloBool	printKeys() const

Inner Class	
IloCsvReader::IloColumnHeaderNotFoundException	Exception thrown for unfound header.
IloCsvReader::IloCsvReaderParameterException	Exception thrown for incorrect arguments in constructor.
IloCsvReader::IloDuplicatedTableException	Exception thrown for tables of same name in csv file.
IloCsvReader::IloFieldNotFoundException	Exception thrown for field not found.
IloCsvReader::IloFileNotFoundException	Exception thrown when file is not found.
IloCsvReader::IloIncorrectCsvReaderUseException	Exception thrown for call to inappropriate csv reader.
IloCsvReader::IloLineNotFoundException	Exception thrown for unfound line.
IloCsvReader::IloTableNotFoundException	Exception thrown for unfound table.
IloCsvReader::LineIterator	Line-iterator for csv readers.
IloCsvReader::TableIterator	Table-iterator of csv readers.

Constructors and Destructors

public **IloCsvReader**()

This constructor creates a csv reader object whose handle pointer is null. This object must be assigned before it can be used.

```
public IloCsvReader(IloCsvReaderI * impl)
```

This constructor creates a handle object (an instance of `IloCsvReader`) from a pointer to an implementation object (an instance of the class `IloCsvReaderI`).

```
public IloCsvReader(const IloCsvReader & csv)
```

This copy constructor creates a handle from a reference to a csv reader object. Both the csv reader object and `csv` point to the same implementation object.

```
public IloCsvReader(IloEnv env, const char * problem, IloBool multiTable=IloFalse,
IloBool allowTableSplitting=IloFalse, const char * separator=",;\t", const char
decimalp='.', const char quote='\"', const char comment='#')
```

This constructor creates a csv reader object for the file `problem` in the environment `env`. If the argument `isCached` has the value `IloTrue`, the data of the file will be stored in the memory.

The cached mode is useful only if you need to read lines by keys. It needs consequent memory consumption and takes time to load data according to the csv file size.

If the argument `isMultiTable` has the value `IloTrue`, the file `problem` is read as a multitable file. The default value is `IloFalse`.

If the argument `allowTableSplitting` has the value `IloFalse`, splitting the table into several parts in the file is not permitted. The default value is `IloFalse`.

The string `separator` represents the characters used as separator in the data file. The default values are `, ; t`.

The character `decimal` represents the character used to write decimal numbers in the data file. The default value is `.` (period).

The character `quote` represents the character used to quote expressions.

The character `comment` represents the character used at the beginning of each commented line. The default value is `#`.

Methods

```
public void end()
```

This member function deallocates the memory used by the csv reader. If you no longer need a csv reader, you can reduce memory consumption by calling this member function.

```
public IloNum getCsvFormat()
```

This member function returns the format of the csv data file. This format is identified in the data file by `ILOG_CSV_FORMAT`.

Example

```
ILOG_CSV_FORMAT;1
```

`getCsvFormat()` returns `1`.

Note

This member function can be used only if `isMultiTable` has the value `IloTrue`.

```
public IloCsvLine getCurrentLine() const
```

This member function returns the last line read by `getLineByKey` or `getLineByNumber`.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

```
public IloEnv getEnv() const
```

This member function returns the environment object corresponding to the invoking csv reader.

```
public IloNum getFileVersion()
```

This member function returns the version of the csv data file. This information is identified in the data file by `ILOG_DATA_SCHEMA`.

Example

```
ILOG_DATA_SCHEMA;PROJECTNAME;0.9
```

`getFileVersion()` returns `0.9`.

Note

This member function can be used only if `isMultiTable` has the value `IloTrue`.

```
public IloCsvReaderI * getImpl() const
```

This member function returns a pointer to the implementation object corresponding to the invoking csv reader.

```
public IloCsvLine getLineByKey(IloInt numberOfKeys, const char *, ...)
```

This member function takes `numberOfKeys` arguments; these arguments are used as one key to identify a line. It returns an instance of `IloCsvLine` representing the line having `(key1, key2, ...)` in the data file. If the number of keys specified is less than the number of keys in the table, this member function throws an exception. Each time `getLineByNumber` or `getLineByKey` is called, the previous line read by one of these methods is deleted.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

`public IloCsvLine` **`getLineByNumber`**`(IloInt i)`

This member function returns an instance of `IloCsvLine` representing the line numbered `i` in the data file. If `i` does not exist, this member function throws an exception. Each time `getLineByNumber` or `getLineByKey` is called, the previous line read by one of these methods is deleted.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

`public IloInt` **`getNumberOfColumns`**`()`

This member function returns the number of columns in the table. If the first column contains the name of the table it is ignored.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

`public IloInt` **`getNumberOfItems`**`()`

This member function returns the number of lines of the table excluding blank lines, commented lines, and the header line.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

`public IloInt` **`getNumberOfKeys`**`() const`

This member function returns the number of keys for the table.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

`public IloInt` **`getNumberOfTables`**`()`

This member function returns the number of tables in the data file.

`public IloInt` **`getPosition`**`(const char * headingName) const`

This member function returns the position (column number) of the `headingName` in the file.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

```
public IloCsvTableReader getReaderForUniqueTableFile() const
```

This member function returns an `IloCsvTableReader` for the unique table contained in the csv data file.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

```
public const char * getRequiredBy()
```

This member function returns the name of the project that uses the csv data file. This information is identified in the data file by `ILOG_DATA_SCHEMA`.

Example

```
ILOG_DATA_SCHEMA;PROJECTNAME;0.9
```

`getRequiredBy()` returns `PROJECTNAME`.

Note

This member function can be used only if `isMultiTable` has the value `IloTrue`.

```
public IloCsvTableReader getTable()
```

This member function returns an instance of `IloCsvTableReader` representing the unique table in the data file.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

```
public IloCsvTableReader getTableByName(const char * name)
```

This member function returns an instance of `IloCsvTableReader` representing the table named `name` in the data file.

Note

This member function can be used only if `isMultiTable` has the value `IloTrue`.

```
public IloCsvTableReader getTableByNumber(IloInt i)
```

This member function returns an instance of `IloCsvTableReader` representing the table numbered `i` in the data file.

Note

This member function can be used only if `isMultiTable` has the value `IloTrue`.

```
public IloBool isHeadingExists(const char * headingName) const
```

This member function returns `IloTrue` if the column header `headingName` exists. Otherwise, it returns `IloFalse`.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

```
public void operator=(const IloCsvReader & csv)
```

This operator assigns an address to the handle pointer of the invoking csv reader. This address is the location of the implementation object of the argument `csv`.

After execution of this operator, both the invoking csv reader and `csv` point to the same implementation object.

```
public IloBool printKeys() const
```

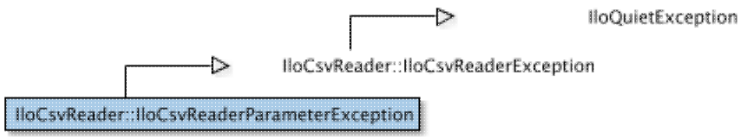
This member function prints the column header of keys if the header exists. Otherwise, it prints the column numbers of keys.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

Class IloCsvReader::IloCsvReaderParameterException

Definition file: ilconcert/ilocsvreader.h



Exception thrown for incorrect arguments in constructor.
This exception is thrown in the constructor of the csv reader if the argument values used in the csv reader constructor are incorrect.

Class IloCsvTableReader

Definition file: ilconcert/ilocsvreader.h

Reads a csv table with format.
An instance of `IloCsvTableReader` is used to read a table of comma-separated values (csv) with a specified format.

An instance is built using a pointer to an implementation class of `IloCsvReader`, which must be created first.

Constructor and Destructor Summary	
public	IloCsvTableReader()
public	IloCsvTableReader(IloCsvTableReaderI * impl)
public	IloCsvTableReader(const IloCsvTableReader & csv)
public	IloCsvTableReader(IloCsvReaderI * csvReaderImpl, const char * name=0)

Method Summary	
public void	end()
public IloCsvLine	getCurrentLine() const
public IloEnv	getEnv() const
public IloCsvTableReaderI *	getImpl() const
public IloCsvLine	getLineByKey(IloInt numberOfKeys, const char *, ...)
public IloCsvLine	getLineByNumber(IloInt i)
public const char *	getNameOfTable() const
public IloInt	getNumberOfColumns()
public IloInt	getNumberOfItems()
public IloInt	getNumberOfKeys() const
public IloInt	getPosition(const char *) const
public IloBool	isHeadingExists(const char * headingName) const
public void	operator=(const IloCsvTableReader & csv)
public IloBool	printKeys() const

Inner Class	
IloCsvTableReader::LineIterator	Line-iterator for csv table readers.

Constructors and Destructors

`public` **`IloCsvTableReader`**`()`

This constructor creates a table csv reader object whose handle pointer is null. This object must be assigned before it can be used.

`public` **`IloCsvTableReader`**`(IloCsvTableReaderI * impl)`

This constructor creates a handle object (an instance of `IloCsvReader`) from a pointer to an implementation object (an instance of the class `IloCsvReaderI`).

```
public IloCsvTableReader(const IloCsvTableReader & csv)
```

This copy constructor creates a handle from a reference to a table csv reader object.

The table csv reader object and `csv` both point to the same implementation object.

```
public IloCsvTableReader(IloCsvReaderI * csvReaderImpl, const char * name=0)
```

This constructor creates a table csv reader object using the implementation class of a csv reader `csvimpl`. The second argument is the name of the table.

Methods

```
public void end()
```

This member function deallocates the memory used by the table csv reader.

If you no longer need the table csv reader, calling this member function can reduce memory consumption.

```
public IloCsvLine getCurrentLine() const
```

This member function returns the last line read using `getLineByKey` or `getLineByNumber`.

```
public IloEnv getEnv() const
```

This member function returns the environment object corresponding to the invoking table csv reader.

```
public IloCsvTableReaderI * getImpl() const
```

This member function returns a pointer to the implementation object corresponding to the invoking table csv reader.

```
public IloCsvLine getLineByKey(IloInt numberOfKeys, const char *, ...)
```

This member function takes `numberOfKeys` arguments. These arguments are used as one key to identify a line. If the specified number of keys is less than the number of keys of the table, this member function throws an exception.

Otherwise, it returns an instance of `IloCsvLine` representing the line having `(key1, key2, ...)` in the data file.

```
public IloCsvLine getLineByNumber(IloInt i)
```

This member function returns an instance of `IloCsvLine` representing the line number `i` in the data file if it exists. Otherwise, it throws an exception.

Each time `getLineByNumber` or `getLineByKey` is called, the previous line read by one of those methods is deleted.

```
public const char * getNameOfTable() const
```

This member function returns the name of the table.

```
public IloInt getNumberOfColumns()
```

This member function returns the number of columns in the table. If the first column contains the name of the table, it is ignored.

```
public IloInt getNumberOfItems()
```

This member function returns the number of lines of the table excluding blank lines, commented lines, and the header line.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

```
public IloInt getNumberOfKeys() const
```

This member function returns the number of keys in the table.

```
public IloInt getPosition(const char *) const
```

This member function returns the position (column number) of `headingName` in the table.

```
public IloBool isHeadingExists(const char * headingName) const
```

This member function returns `IloTrue` if the column header named `headingName` exists. Otherwise, it returns `IloFalse`.

```
public void operator=(const IloCsvTableReader & csv)
```

This operator assigns an address to the handle pointer of the invoking table csv reader.

This address is the location of the implementation object of the argument `csv`.

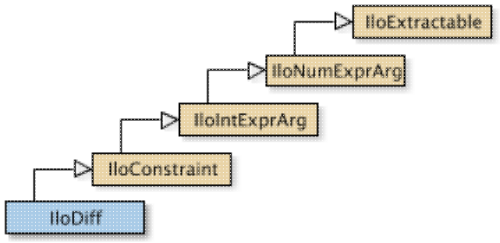
After execution of this operator, the invoking table csv reader and `csv` both point to the same implementation object.

```
public IloBool printKeys() const
```

This member function prints the column headers of keys if they exist. Otherwise, it prints the column numbers of keys.

Class IloDiff

Definition file: ilconcert/ilomodel.h



Constraint that enforces inequality.
An instance of this class is a constraint that enforces inequality (that is, “not equal” as specified by `!=`) in Concert Technology.

To create a constraint, you can:

- use the inequality `operator!=` on constrained variables (instances of `IloNumVar` and its subclasses) or expressions (instances of `IloExpr` and its subclasses).
- use a constructor from this class.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloConstraint, IloExpr, IloNumVar

Constructor Summary	
public	`IloDiff()`
public	`IloDiff(IloDiffI * impl)`
public	`IloDiff(const IloEnv env, const IloNumExprArg expr1, const IloNumExprArg expr2, const char * name=0)`
public	`IloDiff(const IloEnv env, const IloNumExprArg expr1, IloNum val, const char * name=0)`

Method Summary	
`public IloDiffI *`	`getImpl() const`

Inherited Methods from `IloConstraint`
`getImpl`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

```
public IloDiff()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloDiff(IloDiffI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloDiff(const IloEnv env, const IloNumExprArg expr1, const IloNumExprArg
expr2, const char * name=0)
```

This constructor creates a constraint that enforces inequality (!=) in a model between the two expressions that are passed as its arguments. You must use the template `IloAdd` or the member function `IloModel::add` to add this constraint to a model in order for it to be taken into account.

The optional argument `name` is set to `0` by default.

```
public IloDiff(const IloEnv env, const IloNumExprArg expr1, IloNum val, const char
* name=0)
```

This constructor creates a constraint that enforces inequality (!=) in a model between the expression `expr1` and the floating-point value that are passed as its arguments. You must use the template `IloAdd` or the member function `IloModel::add` to add this constraint to a model in order for it to be taken into account.

The optional argument `name` is set to `0` by default.

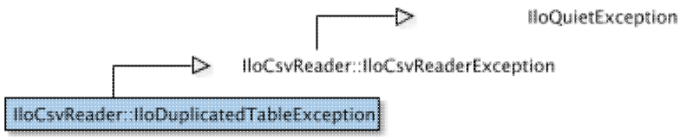
Methods

```
public IloDiffI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

Class IloCsvReader::IloDuplicatedTableException

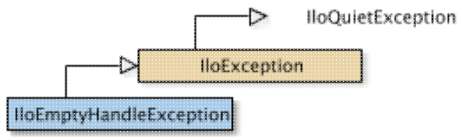
Definition file: ilconcert/ilocsvreader.h



Exception thrown for tables of same name in csv file.
This exception is thrown in the constructor of the csv reader if you read a multitable file in which two tables have the same name but table splitting has not been specified.

Class IloEmptyHandleException

Definition file: ilconcert/iloenv.h



The class of exceptions thrown if an empty handle is passed.
The exception `IloEmptyHandleException` is thrown if an empty handle is passed as an argument to a method, function, or class constructor.

Constructor and Destructor Summary	
`public`	`IloEmptyHandleException()`
`public`	`IloEmptyHandleException(const char * message)`

Inherited Methods from `IloException`
`end, getMessage`

Constructors and Destructors

`public` **`IloEmptyHandleException`**`()`

`public` **`IloEmptyHandleException`**`(const char * message)`

This constructor creates an exception containing the message string `message`.

Class IloEnv

Definition file: ilconcert/iloenv.h

The class of environments for models or algorithms in Concert Technology.
An instance of this class is an environment, managing memory and identifiers for modeling objects. Every Concert Technology object, such as an extractable object, a model, or an algorithm, must belong to an environment. In C++ terms, when you construct a model (an instance of `IloModel`) or an algorithm (an instance of `IloCplex`, `IloCP`, or `IloSolver`, for example), then you must pass one instance of `IloEnv` as an argument of that constructor.

Environment and Memory Management

An environment (an instance of `IloEnv`) efficiently manages memory allocations for the objects constructed with that environment as an argument. For example, when Concert Technology objects in your model are extracted by an algorithm, those extracted objects are handled as efficiently as possible with respect to memory management; there is no unnecessary copying that might cause memory explosions in your application on the part of Concert Technology.

When your application deletes an instance of `IloEnv`, Concert Technology will automatically delete all models and algorithms depending on that environment as well. You delete an environment by calling the member function `env.end`.

The memory allocated for Concert Technology arrays, expressions, sets, and columns is not freed until all references to these objects have terminated and the objects themselves have been deleted.

Certain classes documented in this manual, such as `IloFastMutex`, are known as system classes. They do not belong to a Concert Technology environment; in other words, an instance of `IloEnv` is *not* an argument in their constructors. As a consequence, a Concert Technology environment does *not* attempt to manage their memory allocation and de-allocation; a call of `IloEnv:end` will *not* delete an instance of a system class. These system classes are clearly designated in this documentation, and the appropriate constructors and destructors for them are documented in this manual as well.

Environment and Initialization

An instance of `IloEnv` in your application initializes certain data structures and modeling facilities for Concert Technology. For example, `IloEnv` initializes the symbolic constant `IloInfinity`.

The environment also specifies the current assumptions about normalization or the reduction of terms in linear expressions. For an explanation of this concept, see the concept Normalization: Reducing Linear Terms

Environment and Communication Streams

An instance of `IloEnv` in your application initializes the default output streams for general information, for error messages, and for warnings.

Environment and Extractable Objects

Every extractable object in your problem must belong to an instance of `IloEnv`. In C++ terms, in the constructor of certain extractable objects that you create, such as a constrained variable, you must pass an instance of `IloEnv` as an argument to specify which environment the extractable object belongs to. An extractable object (that is, an instance of `IloExtractable` or one of its derived subclasses) is tied throughout its lifetime to the environment where it is created. It can be used only with extractable objects belonging to the same environment. It can be extracted only for an algorithm attached to the same environment.

Two different environments cannot share the same extractable object.

You can extract objects from only one environment into a given algorithm. In other words, algorithms do not extract objects from two or more different environments.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloException, IloModel

Constructor Summary	
public	IloEnv()
public	IloEnv(IloEnvI * impl)

Method Summary	
public void	end()
public ostream &	error() const
public IloExtractableI *	getExtractable(IloInt id)
public IloEnvI *	getImpl() const
public IloInt	getMaxId() const
public IloInt	getMemoryUsage() const
public ostream &	getNullStream() const
public IloRandom	getRandom() const
public IloNum	getTime() const
public IloInt	getTotalMemoryUsage() const
public const char *	getVersion() const
public IloBool	isValidId(IloInt id) const
public ostream &	out() const
public void	printTime() const
public void	setError(ostream & s)
public void	setNormalizer(IloBool val) const
public void	setOut(ostream & s)
public void	setWarning(ostream & s)
public ostream &	warning() const

Constructors

public **IloEnv**()

This constructor creates an environment to manage the extractable objects in Concert Technology.

public **IloEnv**(IloEnvI * impl)

This constructor creates an environment (a handle) from its implementation object.

Methods

```
public void end()
```

When you call this member function, it cleans up the invoking environment. In other words, it deletes all the extractable objects (instances of `IloExtractable` and its subclasses) created in that environment and frees the memory allocated for them. It also deletes all algorithms (instances of `IloAlgorithm` and its subclasses) created in that environment and frees memory allocated for them as well, including the representations of extractable objects extracted for those algorithms.

```
public ostream & error() const
```

This member function returns a reference to the output stream currently used for error messages from the invoking environment. It is initialized as `cerr`.

```
public IloExtractableI * getExtractable(IloInt id)
```

This member function returns the extractable associated with the specified identifier `id`.

```
public IloEnvI * getImpl() const
```

This member function returns the implementation object of the invoking environment.

```
public IloInt getMaxId() const
```

This member function returns the highest id of all extractables int the current `IloEnv`

```
public IloInt getMemoryUsage() const
```

This member function returns a value in bytes specifying how full the heap is.

```
public ostream & getNullStream() const
```

This member function calls the null stream of the environment. This member function can be used with `IloAlgorithm::setOut()` to suppress screen output by redirecting it to the null stream.

```
public IloRandom getRandom() const
```

Each instance of `IloEnv` contains a random number generator, an instance of the class `IloRandom`. This member function returns that `IloRandom` instance.

```
public IloNum getTime() const
```

This member function returns the amount of time elapsed in seconds since the construction of the invoking environment. (The member function `IloEnv::printTime` directs this information to the output stream of the

invoking environment.)

```
public IloInt getTotalMemoryUsage() const
```

This member function returns a value in bytes specifying how large the heap is.

```
public const char * getVersion() const
```

This member function returns a string specifying the version of IBM ILOG Concert Technology.

```
public IloBool isValidId(IloInt id) const
```

This methods tells you if the current `id` is associated with a live extractable.

```
public ostream & out() const
```

This member function returns a reference to the output stream currently used for logging. General output from the invoking environment is accessible through this member function. By default, the logging output stream is defined by an instance of `IloEnv` as `cout`.

```
public void printTime() const
```

This member function directs the output of the member function `IloEnv::getTime` to the output stream of the invoking environment. (The member function `IloEnv::getTime` accesses the elapsed time in seconds since the creation of the invoking environment.)

```
public void setError(ostream & s)
```

This member function sets the stream for errors generated by the invoking environment. By default, the stream is defined by an instance of `IloEnv` as `cerr`.

```
public void setNormalizer(IloBool val) const
```

This member function turns on or off the facilities in Concert Technology for normalizing linear expressions. Normalizing linear expressions is also known as reducing the terms of a linear expression. In this context, a linear expression that does not contain multiple terms with the same variable is said to be normalized. The concept in this manual offers examples of this idea.

When `val` is `IloTrue`, (the default), then Concert Technology analyzes linear expressions to determine whether any variable appears more than once in a given linear expression. It then combines terms in the linear expression to eliminate any duplication of variables. This mode may require more time during preliminary computation, but it avoids the possibility of an assertion failing in the case of duplicated variables in the terms of a linear expression.

When `val` is `IloFalse`, then Concert Technology assumes that all linear expressions in the invoking environment have already been processed to reduce them to their most efficient form. In other words, Concert Technology assumes that linear expressions have been normalized. This mode may save time during computation, but it entails the risk that a linear expression may contain one or more variables, each of which

appears in one or more terms. This situation will cause certain `assert` statements in Concert Technology to fail if you do not compile with the flag `-DNDEBUG`.

```
public void setOut(ostream & s)
```

This member function redirects the `out()` stream with the stream given as an argument.

This member function can be used with `IloEnv::getNullStream` to suppress screen output by redirecting it to the null stream.

```
public void setWarning(ostream & s)
```

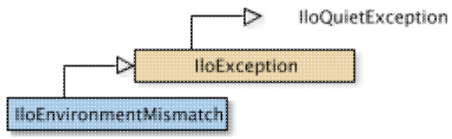
This member function sets the stream for warnings from the invoking environment. By default, the stream is defined by an instance of `IloEnv` as `cout`.

```
public ostream & warning() const
```

This member function returns a reference to the output stream currently used for warnings from the invoking environment. By default, the warning output stream is defined by an instance of `IloEnv` as `cout`.

Class IloEnvironmentMismatch

Definition file: ilconcert/iloenv.h



This exception is thrown if you try to build an object using objects from another environment.
The `IloEnvironmentMismatch` exception is thrown if you try to build an object using objects from another environment.

Constructor and Destructor Summary	
public	IloEnvironmentMismatch()
public	IloEnvironmentMismatch(const char * message)
public	~IloEnvironmentMismatch()

Inherited Methods from `IloException`
end, getMessage

Constructors and Destructors

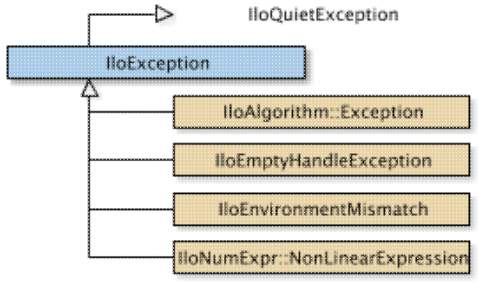
public **IloEnvironmentMismatch**()

public **IloEnvironmentMismatch**(const char * message)

public **~IloEnvironmentMismatch**()

Class IloException

Definition file: ilconcert/ilosys.h



Base class of Concert Technology exceptions.
This class is the base class for exceptions in Concert Technology. An instance of this class represents an exception on platforms that support exceptions when exceptions are enabled.

See Also: IloEnv, operator<<

Constructor and Destructor Summary	
protected	IloException(const char * message=0, IloBool deleteMessage=IloFalse)

Method Summary	
public virtual void	end()
public virtual const char *	getMessage() const

Constructors and Destructors

```
protected IloException(const char * message=0, IloBool deleteMessage=IloFalse)
```

This protected constructor creates an exception.

Methods

```
public virtual void end()
```

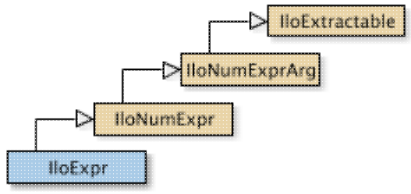
This member function deletes the invoking exception. That is, it frees memory associated with the invoking exception.

```
public virtual const char * getMessage() const
```

This member function returns the message (a character string) of the invoking exception.

Class IloExpr

Definition file: ilconcert/iloexpression.h



An instance of this class represents an expression in a model.
An instance of `IloExpr` is a handle.

Expressions in Environments

The variables in an expression must all belong to the same environment as the expression itself. In other words, you must not mix variables from different environments within the same expression.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Programming Hint: Creating Expressions

In addition to using a constructor of this class to create an expression, you may also initialize an instance of `IloExpr` as a C++ expression built from variables of a model. For example:

```
IloNumVar x;
IloNumVar y;
IloExpr expr = x + y;
```

Programming Hint: Empty Handles and Null Expressions

This statement creates an empty handle:

```
IloExpr e1;
```

You must initialize it before you use it. For example, if you attempt to use it in this way:

```
e1 += 10; // BAD IDEA
```

Without the compiler option `-DNDEBUG`, that line will cause an `assert` statement to fail because you are attempting to use an empty handle.

In contrast, the following statement

```
IloExpr e2(env);
```

creates a handle to a null expression. You can use this handle to build up an expression, for example, in this way:

```
e2 += 10; // OK
```

Normalizing Linear Expressions: Reducing the Terms

Normalizing is sometimes known as *reducing the terms* of a linear expression.

Linear expressions consist of terms made up of constants and variables related by arithmetic operations; for example, x + 3y is a linear expression of two terms consisting of two variables. In some expressions, a given variable may appear in more than one term, for example, x + 3y +2x. Concert Technology has more than one

way of dealing with linear expressions in this respect, and you control which way Concert Technology treats expressions from your application.

In one mode, Concert Technology analyzes linear expressions that your application passes it and attempts to reduce them so that a given variable appears in only one term in the linear expression. This is the default mode. You set this mode with the member function `IloEnv::setNormalizer(IloTrue)`.

In the other mode, Concert Technology assumes that no variable appears in more than one term in any of the linear expressions that your application passes to Concert Technology. We call this mode assume normalized linear expressions. You set this mode with the member function `IloEnv::setNormalizer(IloFalse)`.

Certain constructors and member functions in this class check this setting in the environment and behave accordingly: they assume that no variable appears in more than one term in a linear expression. This mode may save time during computation, but it entails the risk that a linear expression may contain one or more variables, each of which appears in one or more terms. Such a case may cause certain assertions in member functions of this class to fail if you do not compile with the flag `-DNDEBUG`.

Certain constructors and member functions in this class check this setting in the environment and behave accordingly: they attempt to reduce expressions. This mode may require more time during preliminary computation, but it avoids of the possibility of a failed assertion in case of duplicates.

See Also: IloExprArray, IloModel

Constructor Summary	
public	IloExpr()
public	IloExpr(IloNumExprI * expr)
public	IloExpr(const IloNumLinExprTerm term)
public	IloExpr(const IloIntLinExprTerm term)
public	IloExpr(IloNumExprArg)
public	IloExpr(const IloEnv env, IloNum=0)

Method Summary	
public IloNum	getConstant() const
public IloNumLinTermI *	getImpl() const
public IloExpr::LinearIterator	getLinearIterator() const
public IloBool	isNormalized() const
public IloInt	normalize() const
public IloExpr &	operator*=(IloNum val)
public IloExpr &	operator+=(const IloIntLinExprTerm term)
public IloExpr &	operator+=(const IloNumLinExprTerm term)
public IloExpr &	operator+=(const IloIntVar var)
public IloExpr &	operator+=(const IloNumVar var)
public IloExpr &	operator+=(const IloNumExprArg expr)
public IloExpr &	operator+=(IloNum val)
public IloExpr &	operator-=(const IloIntLinExprTerm term)
public IloExpr &	operator-=(const IloNumLinExprTerm term)
public IloExpr &	operator-=(const IloIntVar var)
public IloExpr &	operator-=(const IloNumVar var)
public IloExpr &	operator-=(const IloNumExprArg expr)

public IloExpr &	operator-=(IloNum val)
public IloExpr &	operator/=(IloNum val)
public void	remove(const IloNumVarArray vars)
public void	setConstant(IloNum cst)
public void	setLinearCoef(const IloNumVar var, IloNum value)
public void	setLinearCoefs(const IloNumVarArray vars, IloNumArray values)
public void	setNumConstant(IloNum constant)

Inherited Methods from **IloNumExpr**
getImpl, operator*=, operator+=, operator+=, operator-=, operator-=, operator/=

Inherited Methods from **IloNumExprArg**
getImpl

Inherited Methods from **IloExtractable**
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Inner Class	
IloExpr::LinearIterator	An iterator over the linear part of an expression.

Constructors

public **IloExpr**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloExpr**(IloNumExprI * expr)

This constructor creates an expression from a pointer to the implementation class of numeric expressions IloNumExprI*.

public **IloExpr**(const IloNumLinExprTerm term)

This constructor creates an integer expression with linear terms using the undocumented class IloNumLinExprTerm.

public **IloExpr**(const IloIntLinExprTerm term)

This constructor creates an integer expression with linear terms using the undocumented class IloIntLinExprTerm.

public **IloExpr**(IloNumExprArg)

This constructor creates an expression using the undocumented class `IloNumExprArg`.

```
public IloExpr(const IloEnv env, IloNum=0)
```

This constructor creates an expression in the environment specified by `env`. It may be used to build other expressions from variables belonging to `env`. You must not mix variables of different environments within an expression.

Methods

```
public IloNum getConstant() const
```

This member function returns the constant term in the invoking expression.

```
public IloNumLinTermI * getImpl() const
```

This member function returns the implementation object of the invoking enumerated variable.

```
public IloExpr::LinearIterator getLinearIterator() const
```

This methods returns a linear iterator on the invoking expression.

```
public IloBool isNormalized() const
```

This member function returns `IloTrue` if the invoking expression has been normalized using `IloExpr::normalize`.

```
public IloInt normalize() const
```

This member function normalizes the invoking linear expression. Normalizing is sometimes known as reducing the terms of a linear expression. That is, if there is more than one linear term using the same variable in the invoking linear expression, then this member function merges those linear terms into a single term expressed in that variable. The return value specifies the number of merged terms.

For example, `1*x + 17*y - 3*x` becomes `17*y - 2*x`, and the member function returns 1 (one).

If you attempt to use this member function on a nonlinear expression, it throws an exception.

```
public IloExpr & operator*=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x * ...`

```
public IloExpr & operator+=(const IloIntLinExprTerm term)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x + ...

```
public IloExpr & operator+=(const IloNumLinExprTerm term)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x + ...

```
public IloExpr & operator+=(const IloIntVar var)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x + ...

```
public IloExpr & operator+=(const IloNumVar var)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x + ...

```
public IloExpr & operator+=(const IloNumExprArg expr)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x + ...

```
public IloExpr & operator+=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x + ...

```
public IloExpr & operator-=(const IloIntLinExprTerm term)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x - ...

```
public IloExpr & operator-=(const IloNumLinExprTerm term)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x - ...

```
public IloExpr & operator-=(const IloIntVar var)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x - ...

```
public IloExpr & operator-=(const IloNumVar var)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`

```
public IloExpr & operator-=(const IloNumExprArg expr)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`

```
public IloExpr & operator-=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`

```
public IloExpr & operator/=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x / ...`

```
public void remove(const IloNumVarArray vars)
```

This member function removes all occurrences of all variables listed in the array `vars` from the invoking expression. For linear expressions, the effect of this member function is equivalent to setting the coefficient for all the variables listed in `vars` to 0 (zero).

```
public void setConstant(IloNum cst)
```

This member function assigns `cst` as the constant term in the invoking expression.

```
public void setLinearCoef(const IloNumVar var, IloNum value)
```

This member function assigns `value` as the coefficient of `var` in the invoking expression if the invoking expression is linear. This member function applies only to linear expressions. In other words, you can not use this member function to change the coefficient of a non linear expression. An attempt to do so will cause Concert Technology to throw an exception.

```
public void setLinearCoefs(const IloNumVarArray vars, IloNumArray values)
```

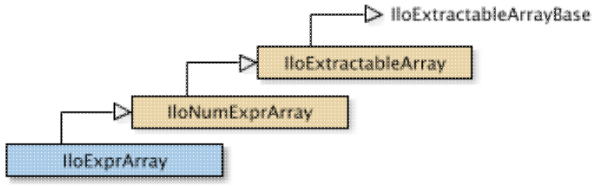
For each of the variables in `vars`, this member function assigns the corresponding value of `values` as its linear coefficient if the invoking expression is linear. This member function applies only to linear expressions. In other words, you can not use this member function to change the coefficient of a nonlinear expression. An attempt to do so will cause Concert Technology to throw an exception.

```
public void setNumConstant(IloNum constant)
```

This member function assigns `cst` as the constant term in the invoking expression.

Class IloExprArray

Definition file: ilconcert/iloexpression.h



The array class of the expressions class.
For each basic type, Concert Technology defines a corresponding array class. `IloExprArray` is the array class of the expressions class (`IloExpr`) for a model.

Instances of `IloExprArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloExpr

Constructor Summary	
public	IloExprArray(IloDefaultArrayI * i=0)
public	IloExprArray(const IloEnv env, IloInt n=0)

Method Summary	
public IloNumExprArg	operator[](IloIntExprArg anIntegerExpr) const

Inherited Methods from **IloNumExprArray**
add, add, add, endElements, operator[]

Inherited Methods from **IloExtractableArray**
add, add, add, endElements, setNames

Constructors

public **IloExprArray**(IloDefaultArrayI * i=0)

This constructor creates an empty array of expressions for use in a model. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

public **IloExprArray**(const IloEnv env, IloInt n=0)

This constructor creates an array of `n` elements. Initially, the `n` elements are empty handles.

Methods

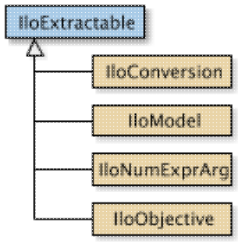
```
public IloNumExprArg operator[](IloIntExprArg anIntegerExpr) const
```

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element expression.

Class IloExtractable

Definition file: ilconcert/iloextractable.h



Base class of all extractable objects.
This class is the base class of all extractable objects (that is, instances of such classes as `IloConstraint`, `IloNumVar`, and so forth). Instances of subclasses of this class represent objects (such as constraints, constrained variables, objectives, and so forth) that can be extracted by Concert Technology from your model for use by your application in Concert Technology algorithms.

Not every algorithm can extract every extractable object of a model. For example, a model may include more than one objective, but you can extract only one objective for an instance of `IloCplex`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Adding Extractable Objects

Generally, for an extractable object to be taken into account by one of the algorithms in Concert Technology, you must add the extractable object to a model with the member function `IloModel::add` and extract the model for the algorithm with the member function `IloAlgorithm::extract`.

Environment and Extractable Objects

Every extractable object in your model must belong to one instance of `IloEnv`. An extractable object (that is, an instance of `IloExtractable` or one of its derived subclasses) is tied throughout its lifetime to the environment where it is created. It can be used only with extractable objects belonging to the same environment. It can be extracted only for an algorithm attached to the same environment.

Notification

When you change an extractable object, for example by removing it from a model, Concert Technology notifies algorithms that have extracted the model containing this extractable object about the change. Member functions that carry out such notification are noted in this documentation.

See Also: IloEnv, IloGetClone, IloModel

Constructor Summary	
public	IloExtractable(IloExtractableI * obj=0)

Method Summary	
public IloConstraint	asConstraint() const
public IloIntExprArg	asIntExpr() const
public IloModel	asModel() const
public IloNumExprArg	asNumExpr() const
public IloObjective	asObjective() const

public IloNumVar	asVariable() const
public void	end()
public IloEnv	getEnv() const
public IloInt	getId() const
public IloExtractableI *	getImpl() const
public const char *	getName() const
public IloAny	getObject() const
public IloBool	isConstraint() const
public IloBool	isIntExpr() const
public IloBool	isModel() const
public IloBool	isNumExpr() const
public IloBool	isObjective() const
public IloBool	isVariable() const
public void	setName(const char * name) const
public void	setObject(IloAny obj) const

Constructors

public **IloExtractable**(IloExtractableI * obj=0)

This constructor creates a handle to the implementation `object`.

Methods

public IloConstraint **asConstraint**() const

This method returns the given extractable as a constraint or a null pointer

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

public IloIntExprArg **asIntExpr**() const

This method returns the given extractable as an integer expression or a null pointer

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

public IloModel **asModel**() const

This method returns the given extractable as a model or a null pointer

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public IloNumExprArg asNumExpr() const
```

This method returns the given extractable as a floating expression or a null pointer

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public IloObjective asObjective() const
```

This method returns the given extractable as an objective or a null pointer

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public IloNumVar asVariable() const
```

This method returns the given extractable as a variable or a null pointer

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public void end()
```

This member function first removes the invoking extractable object from all other extractable objects where it is used (such as a model, ranges, etc.) and then deletes the invoking extractable object. That is, it frees all the resources used by the invoking object. After a call to this member function, you can not use the invoking extractable object again.

Note

The member function `end` notifies Concert Technology algorithms about the destruction of this invoking object.

```
public IloEnv getEnv() const
```

This member function returns the environment to which the invoking extractable object belongs. An extractable object belongs to exactly one environment; different environments can not share the same extractable object.

```
public IloInt getId() const
```

This member function returns the ID of the invoking extractable object.

```
public IloExtractableI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking extractable object. This

member function is useful when you need to be sure that you are using the same copy of the invoking extractable object in more than one situation.

```
public const char * getName() const
```

This member function returns a character string specifying the name of the invoking object (if there is one).

```
public IloAny getObject() const
```

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

```
public IloBool isConstraint() const
```

This method tells you whether or not the given extractable is a constraint or not

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public IloBool isIntExpr() const
```

This method tells you whether or not the given extractable is an integer expression or not

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public IloBool isModel() const
```

This method tells you whether ot not the given extractable is a model or not

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public IloBool isNumExpr() const
```

This method tells you whether or not the given extractable is a floating expression or not

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public IloBool isObjective() const
```

This method tells you whether or not the given extractable is an objective or not

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public IloBool isVariable() const
```

This method tells you whether or not the given extractable is a variable or not

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public void setName(const char * name) const
```

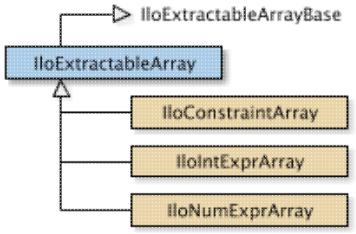
This member function assigns `name` to the invoking object.

```
public void setObject(IloAny obj) const
```

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterward. Normally, `obj` contains user data pertinent to the invoking object.

Class IloExtractableArray

Definition file: ilconcert/iloextractable.h



An array of extractable objects.
An instance of this class is an array of extractable objects (instances of the class `IloExtractable` or its subclasses).

Instances of `IloExtractableArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

For information on arrays, see the concept Arrays

See Also: IloArray, IloExtractable, operator<<

Constructor Summary	
public	IloExtractableArray(IloDefaultArrayI * i=0)
public	IloExtractableArray(const IloExtractableArray & r)
public	IloExtractableArray(const IloEnv env, IloInt n=0)

Method Summary	
public void	add(IloInt more, const IloExtractable x)
public void	add(const IloExtractable x)
public void	add(const IloExtractableArray x)
public void	endElements()
public void	setNames(const char * name)

Constructors

`public` **`IloExtractableArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty array of elements. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

`public` **`IloExtractableArray`**`(const IloExtractableArray & r)`

This copy constructor creates a handle to the array of extractable objects specified by `r`.

```
public IloExtractableArray(const IloEnv env, IloInt n=0)
```

This constructor creates an array of `n` elements, each of which is an empty handle.

Methods

```
public void add(IloInt more, const IloExtractable x)
```

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

```
public void add(const IloExtractable x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloExtractableArray x)
```

This member function appends the elements in the array`x` to the invoking array.

```
public void endElements()
```

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

```
public void setNames(const char * name)
```

This member function set the name for all elements of the invoking array. All elements must be different, otherwise raise an error.

Class IloExtractableVisitor

Definition file: ilconcert/iloextractable.h

The class for inspecting all nodes of an expression.
The class `IloExtractableVisitor` is used to introspect a Concert object and inspect all nodes of the expression.

For example, you can introspect a given expression and look for all the variables within.

You can do this by specializing the `visitChildren` methods and calling the `beginVisit` method on the extractable you want to introspect.

For example, if you visit an `IloDiff` object, you will visit the first expression, then the second expression. When visiting the first or second expression, you will visit their sub-expressions, and so on.

Constructor and Destructor Summary	
`public`	`IloExtractableVisitor()`

Method Summary	
`public virtual void`	`beginVisit(IloExtractableI * e)`
`public virtual void`	`endVisit(IloExtractableI * e)`
`public virtual void`	`visitChildren(IloExtractableI * parent, IloExtractableArray children)`
`public virtual void`	`visitChildren(IloExtractableI * parent, IloExtractableI * child)`

Constructors and Destructors

`public` **`IloExtractableVisitor`**`()`

The default constructor.

Methods

`public virtual void` **`beginVisit`**`(IloExtractableI * e)`

This method begins the introspection.

`public virtual void` **`endVisit`**`(IloExtractableI * e)`

This method ends the inspection.

`public virtual void` **`visitChildren`**`(IloExtractableI * parent, IloExtractableArray children)`

This method is called when the member of the object is an array.

For example, when visiting an `IloAllDiff(env, [x,y,z])` , you use

```
visitChildren(AllDiff, [x,y,z])
```

```
public virtual void visitChildren(IloExtractableI * parent, IloExtractableI *
child)
```

This method is called when visiting a sub-extractable.

For example, if you want to display all the variables in your object, you use:

```
visitChildren(IloExtractableI* parent, IloExtractableI* child){
IloExtractable extr(child); if (child.isVariable()) cout << extr;
}
```

If you visit `IloDiff(env, X, Y)` , for example, you would call this method as:

```
visitChildren(Diff, X)
```

then

```
visitChildren(Diff, Y)
```

Class IloFastMutex

Definition file: ilconcert/ilothread.h

Synchronization primitives adapted to the needs of Concert Technology.
The class `IloFastMutex` provides synchronization primitives adapted to the needs of Concert Technology. In particular, an instance of the class `IloFastMutex` is a nonrecursive mutex that implements mutual exclusion from critical sections of code in multithreaded applications. The purpose of a mutex is to guarantee that concurrent calls to a critical section of code in a multithreaded application are serialized. If a critical section of code is protected by a mutex, then two (or more) threads cannot execute the critical section simultaneously. That is, an instance of this class makes it possible for you to serialize potentially concurrent calls.

Concert Technology implements a mutex by using a single resource that you lock when your application enters the critical section and that you unlock when you leave. Only one thread can own that resource at a given time.

Protection by a Mutex

A critical section of code in a multithreaded application is protected by a mutex when that section of code is encapsulated by a pair of calls to the member functions `IloFastMutex::lock` and `IloFastMutex::unlock`.

In fact, we say that a pair of calls to the member functions `lock` and `unlock` defines a critical section. The conventional way of defining a critical section looks like this:

```
mutex.lock();
while (conditionC does not hold)
       condition.wait(&mutex);
doTreatmentT();
mutex.unlock();
```

The class `IloCondition` provides synchronization primitives to express conditions in critical sections of code.

State of a Mutex

A mutex (an instance of `IloFastMutex`) has a state; the state may be locked or unlocked. You can inquire about the state of a mutex to determine whether it is locked or unlocked by using the member function `IloFastMutex::isLocked`. When a thread enters a critical section of code in a multithreaded application and then locks the mutex defining that critical section, we say that the thread owns that lock and that lock belongs to the thread until the thread unlocks the mutex.

Exceptions

The member functions `IloFastMutex::lock` and `IloFastMutex::unlock` can throw C++ exceptions when exceptions are enabled on platforms that support them. These are the possible exceptions:

- `IloMutexDeadlock`: Instances of `IloFastMutex` are not recursive. Consequently, if a thread locks a mutex and then attempts to lock that mutex again, the member function `lock` throws the exception `MutexDeadlock`. On platforms that do not support exceptions, it causes the application to exit.
- `IloMutexNotOwner`: The thread that releases a given lock (that is, the thread that unlocks a mutex) must be the same thread that locked the mutex in the first place. For example, if a thread `A` takes lock `L` and thread `B` attempts to unlock `L`, then the member function `unlock` throws the exception `MutexNotOwner`. On platforms that do not support exceptions, it causes the application to exit.
- `IloMutexNotOwner`: The member function `unlock` throws this exception whenever a thread attempts to unlock an instance of `IloFastMutex` that is not already locked. On platforms that do not support exceptions, it causes the application to exit.

System Class: Memory Management

`IloFastMutex` is a system class.

Most Concert Technology classes are actually handle classes whose instances point to objects of a corresponding implementation class. For example, instances of the Concert Technology class `IloNumVar` are handles pointing to instances of the implementation class `IloNumVarI`. Their allocation and de-allocation in internal data structures of Concert Technology are managed by an instance of `IloEnv`.

However, system classes, such as `IloFastMutex`, differ from that pattern. `IloFastMutex` is an ordinary C++ class. Its instances are allocated on the C++ heap.

Instances of `IloFastMutex` are not automatically de-allocated by a call to `IloEnv::end`. You must explicitly destroy instances of `IloFastMutex` by means of a call to the delete operator (which calls the appropriate destructor) when your application no longer needs instances of this class.

Furthermore, you should not allocate—neither directly nor indirectly—any instance of `IloFastMutex` in the Concert Technology environment because the destructor for that instance of `IloFastMutex` will never be called automatically by `IloEnv::end` when it cleans up other Concert Technology objects in the Concert Technology environment. In other words, allocation of any instance of `IloFastMutex` in the Concert Technology environment will produce memory leaks.

For example, it is not a good idea to make an instance of `IloFastMutex` part of a conventional Concert Technology model allocated in the Concert Technology environment because that instance will not automatically be de-allocated from the Concert Technology environment along with the other Concert Technology objects.

De-allocating Instances of IloFastMutex

Instances of `IloFastMutex` differ from the usual Concert Technology objects because they are not allocated in the Concert Technology environment, and their de-allocation is not managed automatically for you by `IloEnv::end`. Instead, you must explicitly destroy instances of `IloFastMutex` by calling the delete operator when your application no longer needs those objects.

See Also: IloBarrier, IloCondition

Constructor and Destructor Summary	
`public`	`IloFastMutex()`
`public`	`~IloFastMutex()`

Method Summary	
`public int`	`isLocked()`
`public void`	`lock()`
`public void`	`unlock()`

Constructors and Destructors

`public` **`IloFastMutex`**`()`

This constructor creates an instance of `IloFastMutex` and allocates it on the C++ heap (not in the Concert Technology environment). This mutex contains operating system-specific resources to represent a lock. You may use this mutex for purposes that are private to a process. Its behavior is undefined for inter-process locking.

`public` **`~IloFastMutex`**`()`

The delete operator calls this destructor to de-allocate an instance of `IloFastMutex`. This destructor is called automatically by the runtime system. The destructor releases operating system-specific resources of the invoking mutex.

Methods

```
public int isLocked()
```

This member function returns a Boolean value that shows the state of the invoking mutex. That is, it tells you whether the mutex is locked by the calling thread (`0`) or unlocked (`1`) or locked by a thread other than the calling thread (also `1`).

```
public void lock()
```

This member function acquires a lock for the invoking mutex on behalf of the calling thread. That lock belongs to the calling thread until the member function `unlock` is called.

If you call this member function and the invoking mutex has already been locked, then the calling thread is suspended until the first lock is released.

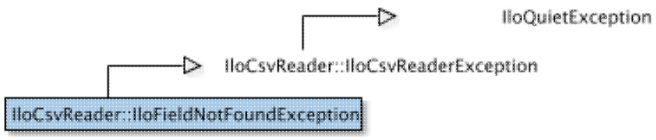
```
public void unlock()
```

This member function releases the lock on the invoking mutex, if there is such a lock.

If you call this member function on a mutex that has not been locked, then this member function throws an exception if C++ exceptions have been enabled on a platform that supports exceptions. Otherwise, it causes the application to exit.

Class IloCsvReader::IloFieldNotFoundException

Definition file: ilconcert/ilocsvreader.h



Exception thrown for field not found.
This exception is thrown by the `IloCsvLine` methods listed below if the corresponding field does not exist.

- IloCsvLine::getFloatByPosition
- IloCsvLine::getIntByPosition
- IloCsvLine::getStringByPosition
- IloCsvLine::getFloatByHeader
- IloCsvLine::getIntByHeader
- IloCsvLine::getStringByHeader
- IloCsvLine::getFloatByPositionOrDefaultValue
- IloCsvLine::getIntByPositionOrDefaultValue
- IloCsvLine::getStringByPositionOrDefaultValue
- IloCsvLine::getFloatByHeaderOrDefaultValue
- IloCsvLine::getIntByHeaderOrDefaultValue
- IloCsvLine::getStringByHeaderOrDefaultValue

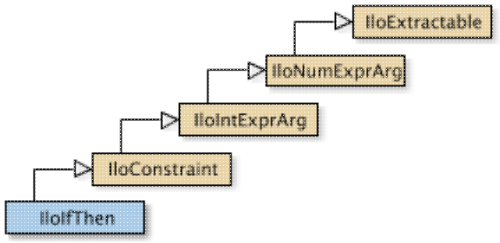
Class IloCsvReader::IloFileNotFoundException

Definition file: ilconcert/ilocsvreader.h

Exception thrown when file is not found.
This exception is thrown in the constructor of the csv reader if a specified file is not found.

Class IloIfThen

Definition file: ilconcert/ilomodel.h



This class represents a condition constraint.
An instance of `IloIfThen` represents a condition constraint. Generally, a condition constraint is composed of an if part (the conditional statement or left side) and a then part (the consequence or right side).

In order for a constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloConstraint

Constructor Summary	
public	`IloIfThen()`
public	`IloIfThen(IloIfThenI * impl)`
public	`IloIfThen(const IloEnv env, const IloConstraint left, const IloConstraint right, const char * name=0)`

Method Summary	
`public IloIfThenI *`	`getImpl() const`

Inherited Methods from `IloConstraint`
`getImpl`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

```
public IloIfThen()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloIfThen(IloIfThenI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloIfThen(const IloEnv env, const IloConstraint left, const IloConstraint
right, const char * name=0)
```

This constructor creates a condition constraint in the environment specified by `env`. The argument `left` specifies the if-part of the condition. The argument `right` specifies the then-part of the condition. The string `name` specifies the name of the constraint; it is `0` (zero) by default. For the constraint to take effect, you must add it to a model and extract the model for an algorithm.

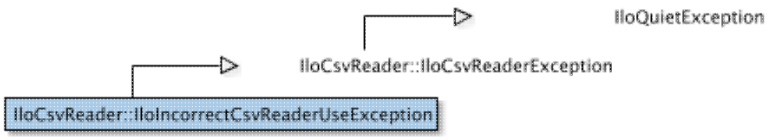
Methods

```
public IloIfThenI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

Class IloCsvReader::IloIncorrectCsvReaderUseException

Definition file: ilconcert/ilocsvreader.h



Exception thrown for call to inappropriate csv reader.
This exception is thrown in the following member functions if you call them from a reader built as a multitable csv reader.

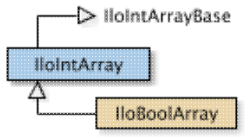
- IloCsvReader::getLineByNumber
- IloCsvReader::getLineByKey
- IloCsvReader::getNumberOfItems
- IloCsvReader::getNumberOfColumns
- IloCsvReader::getNumberOfKeys
- IloCsvReader::getReaderForUniqueTableFile
- IloCsvReader::getTable
- IloCsvReader::isHeadingExists
- IloCsvReader::printKeys

This exception is throw in the following member functions if you call them from a reader built as a unique table csv reader.

- IloCsvReader::getCsvFormat
- IloCsvReader::getFileVersion
- IloCsvReader::getTableByName
- IloCsvReader::getTableByNumber
- IloCsvReader::getRequiredBy

Class IloIntArray

Definition file: ilconcert/iloenv.h



The array class of the basic integer class.
`IloIntArray` is the array class of the basic integer class for a model. It is a handle class. The implementation class for `IloIntArray` is the undocumented class `IloIntArrayI`.

Instances of `IloIntArray` are extensible. (They differ from instances of `IlcIntArray` in this respect.) References to an array change whenever an element is added to or removed from the array.

For each basic type, Concert Technology defines a corresponding array class. That array class is a handle class. In other words, an object of that class contains a pointer to another object allocated in a Concert Technology environment associated with a model. Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object: as a developer using handles, you do not have to worry about memory allocation.

As handles, these objects should be passed by value, and they should be created as automatic objects, where "automatic" has the usual C++ meaning.

Member functions of a handle class correspond to member functions of the same name in the implementation class.

Assert and NDEBUG

Most member functions of the class `IloIntArray` are inline functions that contain an `assert` statement. This statement checks that the handle pointer is not null. These statements can be suppressed by the macro `NDEBUG`. This option usually reduces execution time. The price you pay for this choice is that attempts to access through null pointers are not trapped and usually result in memory faults.

`IloIntArray` inherits additional methods from the template `IloArray`:

- `IloArray::add`
- `IloArray::add`
- `IloArray::clear`
- `IloArray::getEnv`
- `IloArray::getSize`
- `IloArray::remove`
- `IloArray::operator[]`
- `IloArray::operator[]`

See Also: IloInt

Constructor Summary	
`public`	`IloIntArray(IloArrayI * i=0)`
`public`	`IloIntArray(const IloEnv env, IloInt n=0)`
`public`	`IloIntArray(const IloEnv env, IloInt n, IloInt v0, IloInt v1...)`

Method Summary	
`public IloBool`	`contains(IloIntArray ax) const`

public IloBool	contains(IloInt value) const
public void	discard(IloIntArray ax)
public void	discard(IloInt value)
public IloIntExprArg	operator[](IloIntExprArg intExp) const
public IloInt &	operator[](IloInt i)
public const IloInt &	operator[](IloInt i) const
public IloNumArray	toNumArray() const

Constructors

`public` **`IloIntArray`**`(IloArrayI * i=0)`

This constructor creates an array of integers from an implementation object.

`public` **`IloIntArray`**`(const IloEnv env, IloInt n=0)`

This constructor creates an array of `n` integers for use in a model in the environment specified by `env`. By default, its elements are empty handles.

`public` **`IloIntArray`**`(const IloEnv env, IloInt n, IloInt v0, IloInt v1...)`

This constructor creates an array of `n` integers; the elements of the new array take the corresponding values: `v0, v1, ..., v(n-1)`.

Methods

`public IloBool` **`contains`**`(IloIntArray ax) const`

This member function checks whether all the values of `ax` are contained or not.

`public IloBool` **`contains`**`(IloInt value) const`

This member function checks whether the value is contained or not.

`public void` **`discard`**`(IloIntArray ax)`

This member function removes elements from the invoking array. It removes the array `ax`.

`public void` **`discard`**`(IloInt value)`

This member function removes elements from the invoking array. It removes the element.

`public IloIntExprArg` **`operator[]`**`(IloIntExprArg intExp) const`

This subscripting operator returns an expression node for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `intExp` is bound to the value `i`, then the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `intExp`.

This operator is also known as an element expression.

```
public IloInt & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`.

```
public const IloInt & operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

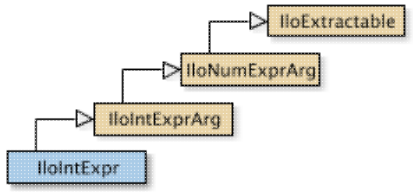
```
 IloArray operator[] (IloInt i) const;
```

```
public IloNumArray toNumArray() const
```

This constructor creates an array of integers from an array of numeric values.

Class IloIntExpr

Definition file: ilconcert/iloexpression.h



The class of integer expressions in Concert Technology.
Integer expressions in Concert Technology are represented using objects of type `IloIntExpr`.

Constructor Summary	
public	IloIntExpr()
public	IloIntExpr(IloIntExprI * impl)
public	IloIntExpr(const IloIntExprArg arg)
public	IloIntExpr(const IloIntLinExprTerm term)
public	IloIntExpr(const IloEnv env, IloInt constant=0)

Method Summary	
public IloIntExprI *	getImpl() const
public IloIntExpr &	operator*=(IloInt val)
public IloIntExpr &	operator+=(const IloIntExprArg expr)
public IloIntExpr &	operator+=(IloInt val)
public IloIntExpr &	operator-=(const IloIntExprArg expr)
public IloIntExpr &	operator-=(IloInt val)

Inherited Methods from **IloIntExprArg**
getImpl

Inherited Methods from **IloNumExprArg**
getImpl

Inherited Methods from **IloExtractable**
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloIntExpr**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloIntExpr**(IloIntExprI * impl)

This constructor creates a handle object from a pointer to an implementation object.

```
public IloIntExpr(const IloIntExprArg arg)
```

This constructor creates an integer expression using the undocumented class `IloIntExprArg`.

```
public IloIntExpr(const IloIntLinExprTerm term)
```

This constructor creates an integer expression with linear terms using the undocumented class `IloIntLinExprTerm`.

```
public IloIntExpr(const IloEnv env, IloInt constant=0)
```

This constructor creates a constant integer expression with the value `constant` that the user can modify subsequently with the operators `+=, -=, =` in the environment `env`.

Methods

```
public IloIntExprI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloIntExpr & operator*=(IloInt val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x * ...`.

```
public IloIntExpr & operator+=(const IloIntExprArg expr)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`.

```
public IloIntExpr & operator+=(IloInt val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`.

```
public IloIntExpr & operator-=(const IloIntExprArg expr)
```

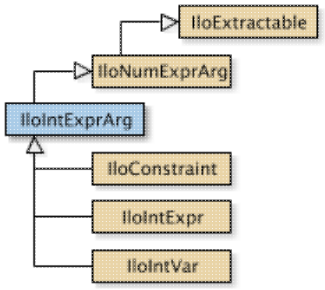
This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`.

```
public IloIntExpr & operator-=(IloInt val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`.

Class IloIntExprArg

Definition file: ilconcert/iloexpression.h



A class used internally in Concert Technology.
Concert Technology uses instances of these classes internally as temporary objects when it is parsing a C++ expression in order to build an instance of `IloIntExpr`. As a Concert Technology user, you will not need this class yourself; in fact, you should not use them directly. They are documented here because the return value of certain functions, such as `IloSum` or `IloScalProd`, can be an instance of this class.

Constructor Summary	
public	IloIntExprArg()
public	IloIntExprArg(IloIntExprI * impl)

Method Summary	
public IloIntExprI *	getImpl() const

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloIntExprArg**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloIntExprArg**(IloIntExprI * impl)

This constructor creates a handle object from a pointer to an implementation object.

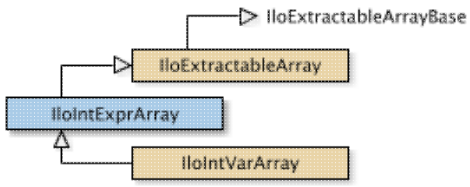
Methods

public IloIntExprI * **getImpl**() const

This member function returns a pointer to the implementation object of the invoking handle.

Class IloIntExprArray

Definition file: ilconcert/iloexpression.h



The array class of the integer expressions class.
For each basic type, Concert Technology defines a corresponding array class. `IloIntExprArray` is the array class of the integer expressions class (`IloIntExpr`) for a model.

Instances of `IloIntExprArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Constructor Summary	
public	`IloIntExprArray(IloDefaultArrayI * i=0)`
public	`IloIntExprArray(const IloEnv env, IloInt n=0)`

Method Summary	
public void	`add(IloInt more, const IloIntExpr x)`
public void	`add(const IloIntExpr x)`
public void	`add(const IloIntExprArray array)`
public void	`endElements()`
public IloIntExprArg	`operator[](IloIntExprArg anIntegerExpr) const`
public IloIntExpr	`operator[](IloInt i) const`
public IloIntExpr &	`operator[](IloInt i)`

Inherited Methods from `IloExtractableArray`
`add, add, add, endElements, setNames`

Constructors

`public` **`IloIntExprArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty array of elements. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

`public` **`IloIntExprArray`**`(const IloEnv env, IloInt n=0)`

This constructor creates an array of n elements. Initially, the n elements are empty handles.

Methods

```
public void add(IloInt more, const IloIntExpr x)
```

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

```
public void add(const IloIntExpr x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloIntExprArray array)
```

This member function appends the elements in `array` to the invoking array.

```
public void endElements()
```

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

```
public IloIntExprArg operator[](IloIntExprArg anIntegerExpr) const
```

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element expression.

```
public IloIntExpr operator[](IloInt i) const
```

This operator returns a reference to the extractable object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

```
 IloIntExpr operator[] (IloInt i) const;
```

```
public IloIntExpr & operator[](IloInt i)
```

This operator returns a reference to the extractable object located in the invoking array at the position specified by the index `i`.

Class IloIntTupleSet

Definition file: ilconcert/ilotupleset.h

Ordered set of values represented by an array.
A tuple is an ordered set of values represented by an array. A *set* of enumerated tuples in a model is represented by an instance of `IloIntTupleSet`. That is, the elements of a tuple *set* are tuples of enumerated values (such as pointers). The number of values in a tuple is known as the *arity* of the tuple, and the arity of the tuples in a set is called the *arity* of the set. (In contrast, the number of tuples in the set is known as the *cardinality* of the set.)

As a handle class, `IloIntTupleSet` manages certain set operations efficiently. In particular, elements can be added to such a set. It is also possible to search a given set with the member function `IloIntTupleSet::isIn` to see whether or not the set contains a given element.

In addition, a set of tuples can represent a constraint defined on a constrained variable, either as the set of *allowed* combinations of values of the constrained variable on which the constraint is defined, or as the set of *forbidden* combinations of values.

There are a few conventions governing tuple sets:

- When you create the set, you must specify the arity of the tuple-elements it contains.
- You use the member function `IloIntTupleSet::add` to add tuples to the set. You can add tuples to the set in a model; you cannot add tuples to an instance of this class during a search, nor inside a constraint, nor inside a goal.

Concert Technology will throw an exception if you attempt:

- to add a tuple with a different number of variables from the arity of the set;
- to search for a tuple with an arity different from the set arity.

See Also: IloIntTupleSetIterator, IloExtractable

Constructor Summary	
public	IloIntTupleSet(const IloEnv env, const IloInt arity)

Method Summary	
public IloBool	add(const IloIntArray tuple) const
public void	end()
public IloInt	getArity() const
public IloInt	getCardinality() const
public IloIntTupleSetI *	getImpl() const
public IloBool	isIn(const IloIntArray tuple) const
public IloBool	remove(const IloIntArray tuple) const

Constructors

`public` **`IloIntTupleSet`**`(const IloEnv env, const IloInt arity)`

This constructor creates a set of tuples (an instance of the class `IloIntTupleSet`) with the arity specified by `arity`.

Methods

```
public IloBool add(const IloIntArray tuple) const
```

This member function adds a tuple represented by the array `tuple` to the invoking set. If you attempt to add an element that is already in the set, that element will *not* be added again. Added elements are not copied; that is, there is no memory duplication. Concert Technology will throw an exception if the size of the array is not equal to the arity of the invoking set. You may use this member function to add tuples to the invoking set in a model; you may not add tuples in this way during a search, inside a constraint, or inside a goal. For those purposes, see `IlcIntTupleSet`, documented in the *IBM ILOG CP Optimizer Reference Manual* and the *IBM ILOG Solver Reference Manual*.

```
public void end()
```

This member function deletes the invoking set. That is, it frees all the resources used by the invoking object. After a call to this member function, you cannot use the invoking extractable object again.

```
public IloInt getArity() const
```

This member function returns the arity of the tupleset.

```
public IloInt getCardinality() const
```

This member function returns the cardinality of the tupleset.

```
public IloIntTupleSetI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking extractable object. This member function is useful when you need to be sure that you are using the same copy of the invoking extractable object in more than one situation.

```
public IloBool isIn(const IloIntArray tuple) const
```

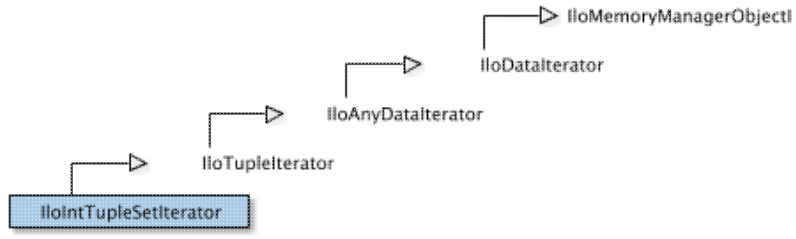
This member function returns `IloTrue` if `tuple` belongs to the invoking set. Otherwise, it returns `IloFalse`. Concert Technology will throw an exception if the size of the array is not equal to the arity of the invoking set.

```
public IloBool remove(const IloIntArray tuple) const
```

This member function removes `tuple` from the invoking set in a model. You may use this member function to remove tuples from the invoking set in a model; you may not remove tuples in this way during a search, inside a constraint, or inside a goal.

Class IloIntTupleSetIterator

Definition file: ilconcert/ilotupleset.h



Class of iterators to traverse enumerated values of a tuple-set.
An instance of the class `IloIntTupleSetIterator` is an iterator that traverses the elements of a finite set of tuples of enumerated values (instance of `IloIntTupleSet`).

See Also the classes `IlcIntTupleSet` in the *IBM ILOG CP Optimizer Reference Manual* and the *IBM ILOG Solver Reference Manual*.

Constructor Summary	
`public`	`IloIntTupleSetIterator(const IloEnv env, IloIntTupleSet tset)`

Method Summary	
`public IloIntArray`	`operator*() const`

Constructors

`public` **`IloIntTupleSetIterator`**`(const IloEnv env, IloIntTupleSet tset)`

This constructor creates an iterator associated with `tSet` to traverse its elements.

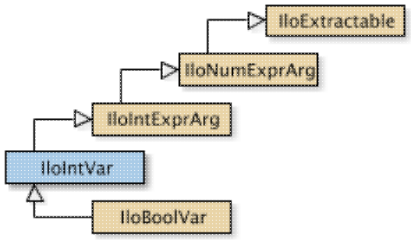
Methods

`public IloIntArray` **`operator*`**`() const`

This operator returns the current element, the one to which the invoking iterator points.

Class IloIntVar

Definition file: ilconcert/iloexpression.h



An instance of this class represents a constrained integer variable in a Concert Technology model.
An instance of this class represents a constrained integer variable in a Concert Technology model. If you are looking for a class of numeric variables that may assume integer values and may be relaxed to assume floating-point values, then consider the class `IloNumVar`. If you are looking for a class of binary decision variables (that is, variables that assume only the values 0 (zero) or 1 (one), then consider the class `IloBoolVar`.

Bounds of an Integer Variable

The lower and upper bound of an instance of this class is an integer.

If you are looking for a symbol to specify an infinite bound, that is, no lower or upper bound, consider `IloIntMin` or `IloIntMax`.

What Is Extracted

An instance of `IloIntVar` is extracted by `IloCP` or `IloSolver` as an instance of `IlcIntVar`.

An instance of `IloIntVar` is extracted by `IloCplex` as a column representing a numeric variable of type `Int` with bounds as specified by `IloIntVar`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Note

When numeric bounds are given to an integer variable (an instance of `IloIntVar` or `IloNumVar` with `Type = Int`) in the constructors or via a modifier (such as `setUB`, `setLB`, `setBounds`), they are inwardly rounded to an integer value. `LB` is rounded down and `UB` is rounded up.

See Also: IloBoolVar, IloNumVar

Constructor Summary	
public	`IloIntVar()`
public	`IloIntVar(IloNumVarI * impl)`
public	`IloIntVar(IloEnv env, IloInt vmin=0, IloInt vmax=IloIntMax, const char * name=0)`
public	`IloIntVar(const IloEnv env, const IloIntArray values, const char * name=0)`
public	`IloIntVar(const IloNumVar var)`

Method Summary	
`public IloNumVarI *`	`getImpl() const`

public IloNum	getLB() const
public IloInt	getMax() const
public IloInt	getMin() const
public IloNum	getUB() const
public void	setBounds(IloInt lb, IloInt ub) const
public void	setLB(IloNum min) const
public void	setMax(IloInt max) const
public void	setMin(IloInt min) const
public void	setPossibleValues(const IloIntArray values) const
public void	setUB(IloNum max) const

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

`public` **`IloIntVar`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloIntVar`**`(IloNumVarI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IloIntVar`**`(IloEnv env, IloInt vmin=0, IloInt vmax=IloIntMax, const char * name=0)`

This constructor creates an instance of `IloIntVar` like this:

```
IloNumVar(env, vmin, vmax, ILOINT, name);
```

`public` **`IloIntVar`**`(const IloEnv env, const IloIntArray values, const char * name=0)`

This constructor calls upon its corresponding `IloNumVar` constructor.

`public` **`IloIntVar`**`(const IloNumVar var)`

This constructor creates a new handle on `var` if it is of type `ILOINT`. Otherwise, an exception is thrown.

Methods

```
public IloNumVarI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getLB() const
```

This member function returns the lower bound of the invoking variable.

```
public IloInt getMax() const
```

This member function returns the maximal value of the invoking variable.

```
public IloInt getMin() const
```

This member function returns the minimal value of the invoking variable.

```
public IloNum getUB() const
```

This member function returns the upper bound of the invoking variable.

```
public void setBounds(IloInt lb, IloInt ub) const
```

This member function sets `lb` as the lower bound and `ub` as the upper bound of the invoking numeric variable.

Note

The member function `setBounds` notifies Concert Technology algorithms about the change of bounds in this numeric variable.

```
public void setLB(IloNum min) const
```

This member function sets `min` as the lower bound of the invoking variable.

Note

The member function `setLB` notifies Concert Technology algorithms about the change of bounds in this numeric variable.

```
public void setMax(IloInt max) const
```

This member function returns the minimal value of the invoking variable to `max`.

Note

The member function `setMax` notifies Concert Technology algorithms about the change of bounds in this numeric variable.

```
public void setMin(IloInt min) const
```

This member function returns the minimal value of the invoking variable to `min`.

Note

The member function `setMin` notifies Concert Technology algorithms about the change of bounds in this numeric variable.

```
public void setPossibleValues(const IloIntArray values) const
```

This member function sets `values` as the domain of the invoking integer variable.

Note

The member function `setPossibleValues` notifies Concert Technology algorithms about the change of bounds in this numeric variable.

```
public void setUB(IloNum max) const
```

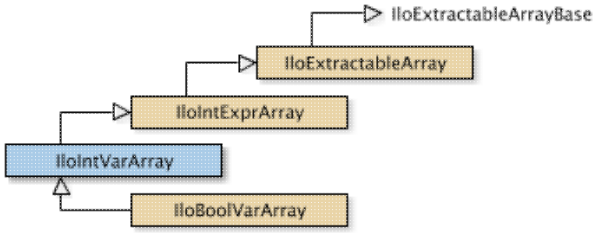
This member function sets `max` as the upper bound of the invoking variable.

Note

The member function `setUB` notifies Concert Technology algorithms about the change of bounds in this numeric variable.

Class IloIntVarArray

Definition file: ilconcert/iloexpression.h



The array class of the integer constrained variables class.
For each basic type, Concert Technology defines a corresponding array class. `IloIntVarArray` is the array class of the integer variable class for a model. It is a handle class.

Instances of `IloIntVarArray` are extensible.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Elements of the array are handles to integer variables. The lower and upper bounds of an integer variable must be an integer. See the documentation of `IloIntVar` for details about bounds of the elements of an array of this class.

See Also: IloIntVar

Constructor Summary	
public	`IloIntVarArray(IloDefaultArrayI * i=0)`
public	`IloIntVarArray(const IloEnv env, IloInt n=0)`
public	`IloIntVarArray(const IloEnv env, const IloIntArray lb, const IloIntArray ub)`
public	`IloIntVarArray(const IloEnv env, IloInt lb, const IloIntArray ub)`
public	`IloIntVarArray(const IloEnv env, const IloIntArray lb, IloInt ub)`
public	`IloIntVarArray(const IloEnv env, IloInt n, IloInt lb, IloInt ub)`
public	`IloIntVarArray(const IloEnv env, const IloNumColumnArray columnarray, const IloNumArray lb, const IloNumArray ub)`

Method Summary	
public void	`add(IloInt more, const IloIntVar x)`
public void	`add(const IloIntVar x)`
public void	`add(const IloIntVarArray x)`
public void	`endElements()`
public IloIntVar	`operator[](IloInt i) const`
public IloIntVar &	`operator[](IloInt i)`
public IloIntExprArg	`operator[](IloIntExprArg anIntegerExpr) const`
public IloNumVarArray	`toNumVarArray() const`

Inherited Methods from `IloIntExprArray`

add, add, add, endElements, operator[], operator[], operator[]

Inherited Methods from `IloExtractableArray`
add, add, add, endElements, setNames

Constructors

`public` **`IloIntVarArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty extensible array of integer variables.

`public` **`IloIntVarArray`**`(const IloEnv env, IloInt n=0)`

This constructor creates an extensible array of `n` integer variables.

`public` **`IloIntVarArray`**`(const IloEnv env, const IloIntArray lb, const IloIntArray ub)`

This constructor creates an extensible array of integer variables with lower and upper bounds as specified.

`public` **`IloIntVarArray`**`(const IloEnv env, IloInt lb, const IloIntArray ub)`

This constructor creates an extensible array of integer variables with a lower bound and an array of upper bounds as specified.

`public` **`IloIntVarArray`**`(const IloEnv env, const IloIntArray lb, IloInt ub)`

This constructor creates an extensible array of integer variables with an array of lower bounds and an upper bound as specified.

`public` **`IloIntVarArray`**`(const IloEnv env, IloInt n, IloInt lb, IloInt ub)`

This constructor creates an extensible array of `n` integer variables, with a lower and an upper bound as specified.

`public` **`IloIntVarArray`**`(const IloEnv env, const IloNumColumnArray columnarray, const IloNumArray lb, const IloNumArray ub)`

This constructor creates an extensible array of integer variables with lower and upper bounds as specified from a column array.

Methods

`public void` **`add`**`(IloInt more, const IloIntVar x)`

This member function appends `x` to the invoking array of integer variables; it appends `x more` times.

```
public void add(const IloIntVar x)
```

This member function appends the value `x` to the invoking array.

```
public void add(const IloIntVarArray x)
```

This member function appends the variables in the array `x` to the invoking array.

```
public void endElements()
```

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

```
public IloIntVar operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator

```
 IloIntVar operator[] (IloInt i) const;
```

```
public IloIntVar & operator[](IloInt i)
```

This operator returns a reference to the extractable object located in the invoking array at the position specified by the index `i`.

```
public IloIntExprArg operator[](IloIntExprArg anIntegerExpr) const
```

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element expression.

```
public IloNumVarArray toNumVarArray() const
```

This member function copies the invoking array into a new `IloNumVarArray`.

Class IloIterator<>

Definition file: ilconcert/iloiterator.h

A template to create iterators for a class of extractable objects.
This template creates iterators for a given class of extractable objects (denoted by `E` in the template) within an instance of `IloEnv`.

By default, an iterator created in this way will traverse instances of `E` and of its subclasses. You can prevent the iterator from traversing instances of subclasses of E (that is, you can limit its effect) by setting the argument `withSubClasses` to `IloFalse` in the constructor of the iterator.

While an iterator created in this way is working, you must not create nor destroy any extractable objects in the instance of `IloEnv` where it is working. In other words, an iterator created in this way works only in a stable environment.

An iterator created with this template differs from an instance of `IloModel::Iterator`. An instance of `IloModel::Iterator` works only on extractable objects (instances of `IloExtractable` or its subclasses) that have explicitly been added to a model (an instance of `IloModel`). In contrast, an iterator created with this template will work on all extractable objects within a given environment, whether or not they have been explicitly added to a model.

See Also: IloEnv, IloExtractable, IloModel, IloModel::Iterator

Constructor Summary	
public	IloIterator(const IloEnv env, IloBool withSubClasses=IloTrue)

Method Summary	
public IloBool	ok()
public void	operator++()

Constructors

`public` **`IloIterator`**`(const IloEnv env, IloBool withSubClasses=IloTrue)`

This template constructor creates an iterator for instances of the class `E`. When the argument `withSubClasses` is `IloTrue` (its default value), the iterator will also work on instances of the subclasses of `E`. When `withSubClasses` is `IloFalse`, the iterator works only on instances of `E`.

Example

Here is an example of an iterator created by this template for the class IloNumVar.

```
typedef IloIterator<IloNumVar> IloNumVarIterator;

void displayAllVars(IloEnv env) {
  for (IloNumVarIterator it(env); it.ok(); ++it) {
    IloNumVar ext = *it;
    cout << ext;
  }
}
```

Methods

`public IloBool` **`ok`**`()`

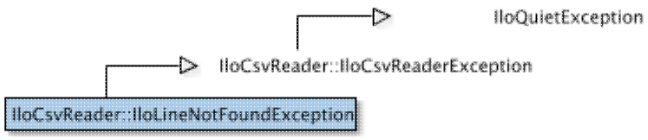
This member function returns `IloTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IloFalse`.

`public void` **`operator++`**`()`

This operator advances the iterator to point to the next value in the iteration.

Class IloCsvReader::IloLineNotFoundException

Definition file: ilconcert/ilocsvreader.h

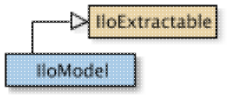


Exception thrown for unfound line.
This exception is thrown by the following member functions if the line is not found.

- IloCsvTableReader::getLineByKey
- IloCsvTableReader::getLineByNumber
- IloCsvReader::getLineByKey
- IloCsvReader::getLineByNumber

Class IloModel

Definition file: ilconcert/ilomodel.h



Class for models.
An instance of this class represents a model. A model consists of the extractable objects such as constraints, constrained variables, objectives, and possibly other modeling objects, that represent a problem. Concert Technology extracts information from a model and passes the information in an appropriate form to algorithms that solve the problem. (For information about extracting objects into algorithms, see the member function `IloAlgorithm::extract` and the template `IloAdd`.)

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Models and Submodels

With Concert Technology, you may create more than one model in a given environment (an instance of `IloEnv`). In fact, you can create submodels. That is, you can add one model to another model within the same environment.

What Is Extracted from a Model

All the extractable objects (that is, instances of `IloExtractable` or one of its subclasses) that have been added to a model (an instance of `IloModel`) and that have not been removed from it will be extracted when an algorithm extracts the model. An instance of the nested class `IloModel::Iterator` accesses those extractable objects.

See Also: IloEnv, IloExtractable, IloModel::Iterator

Constructor Summary	
public	`IloModel()`
public	`IloModel(IloModelI * impl)`
public	`IloModel(const IloEnv env, const char * name=0)`

Method Summary	
public const IloExtractableArray &	`add(const IloExtractableArray & x) const`
public IloExtractable	`add(const IloExtractable x) const`
public IloModelI *	`getImpl() const`
public void	`remove(const IloExtractableArray x) const`
public void	`remove(const IloExtractable x) const`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Inner Class	
IloModel::Iterator	Nested class of iterators to traverse the extractable objects in a model.

Constructors

```
public IloModel()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloModel(IloModelI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloModel(const IloEnv env, const char * name=0)
```

This constructor creates a model. By default, the name of the model is the empty string, but you can attribute a name to the model at its creation.

Methods

```
public const IloExtractableArray & add(const IloExtractableArray & x) const
```

This member function adds the array of extractable objects to the invoking model.

> **Note**
>
> The member function `add` notifies Concert Technology algorithms about this addition to the model.

```
public IloExtractable add(const IloExtractable x) const
```

This member function adds the extractable object to the invoking model.

> **Note**
>
> The member function `add` notifies Concert Technology algorithms about this addition to the model.

```
public IloModelI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void remove(const IloExtractableArray x) const
```

This member function removes the array of extractable objects from the invoking model.

> **Note**
>
> The member function `remove` notifies Concert Technology algorithms about this removal from the model.

```
public void remove(const IloExtractable x) const
```

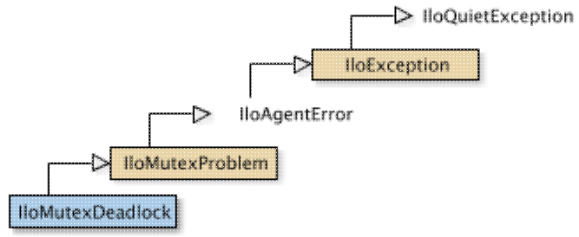
This member function removes the extractable object from the invoking model.

Note

The member function `remove` notifies Concert Technology algorithms about this removal from the model.

Class IloMutexDeadlock

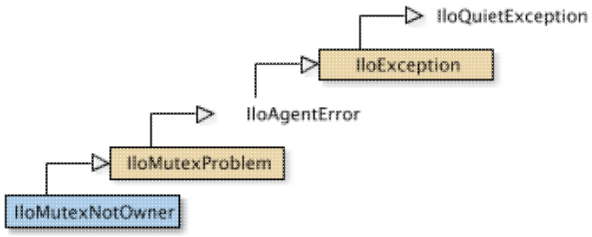
Definition file: ilconcert/ilothread.h



The class of exceptions thrown due to mutex deadlock.
This is the class of exceptions thrown if two or more threads become deadlocked waiting for a mutex owned by the other(s).

Class IloMutexNotOwner

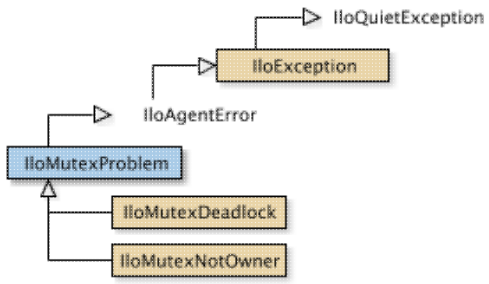
Definition file: ilconcert/ilothread.h



The class of exceptions thrown.
The class of exceptions thrown if a thread attempts to unlock a mutex that it does not own.

Class IloMutexProblem

Definition file: ilconcert/ilothread.h



Exception.
The class `IloMutexProblem` is part of the hierarchy of classes representing exceptions in Concert Technology. Concert Technology uses instances of this class when an error occurs with respect to a mutex, an instance of `IloFastMutex`.

An exception is thrown; it is not allocated in a Concert Technology environment; it is not allocated on the C++ heap. It is not necessary for you as a programmer to delete an exception explicitly. Instead, the system calls the constructor of the exception to create it, and the system calls the destructor of the exception to delete it.

When exceptions are enabled on a platform that supports C++ exceptions, an instance of `IloMutexProblem` makes it possible for Concert Technology to throw an exception in case of error. On platforms that do not support C++ exceptions, an instance of this class makes it possible for Concert Technology to exit in case of error.

Throwing and Catching Exceptions

Exceptions are thrown by value. They are not allocated on the C++ heap, nor in a Concert Technology environment. The correct way to catch an exception is to catch a reference to the error (specified by the ampersand `&`), like this:

```
catch(IloMutexProblem& error);
```

See Also: IloException, IloFastMutex

Constructor Summary	
public	IloMutexProblem(const char * msg)

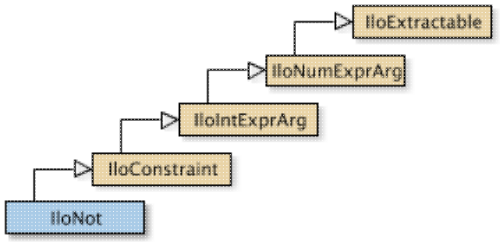
Constructors

`public` **`IloMutexProblem`**`(const char * msg)`

This constructor creates an instance of IloMutexProblem to represent an exception in case of an error involving a mutex. This instance is not allocated on C++ heap; it is not allocated in a Concert Technology environment either.

Class IloNot

Definition file: ilconcert/ilomodel.h



Negation of its argument.
The class `IloNot` represents a constraint that is the negation of its argument. In order to be taken into account, this constraint must be added to a model and extracted by an algorithm, such as `IloCplex` or `IloSolver`.

See Also: operator!

Constructor Summary	
`public`	`IloNot()`
`public`	`IloNot(IloNotI * impl)`

Method Summary	
`public IloNotI *`	`getImpl() const`

Inherited Methods from `IloConstraint`
`getImpl`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

`public` **`IloNot`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloNot`**`(IloNotI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

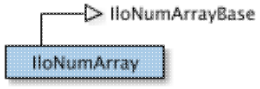
Methods

```
public IloNotI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

Class IloNumArray

Definition file: ilconcert/iloenv.h



The array class of the basic floating-point class.
For each basic type, Concert Technology defines a corresponding array class. `IloNumArray` is the array class of the basic floating-point class (`IloNum`) for a model.

Instances of `IloNumArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

`IloNumArray` inherits additional methods from the template `IloArray`:

- `IloArray::add`
- `IloArray::add`
- `IloArray::add`
- `IloArray::clear`
- `IloArray::getEnv`
- `IloArray::getSize`
- `IloArray::remove`
- `IloArray::operator[]`
- `IloArray::operator[]`

See Also: IloNum, operator>>, operator<<

Constructor Summary	
public	IloNumArray(IloArrayI * i=0)
public	IloNumArray(const IloNumArray & cpy)
public	IloNumArray(const IloEnv env, IloInt n=0)
public	IloNumArray(const IloEnv env, IloInt n, IloNum f0, IloNum f1, ...)

Method Summary	
public IloBool	contains(IloNum value) const
public IloNum &	operator[](IloInt i)
public const IloNum &	operator[](IloInt i) const
public IloNumExprArg	operator[](IloIntExprArg intExp) const
public IloIntArray	toIntArray() const

Constructors

`public` **`IloNumArray`**`(IloArrayI * i=0)`

This constructor creates an empty array of floating-point numbers for use in a model. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

```
public IloNumArray(const IloNumArray & cpy)
```

This copy constructor creates a handle to the array of floating-point objects specified by `cpy`.

```
public IloNumArray(const IloEnv env, IloInt n=0)
```

This constructor creates an array of `n` elements. Initially, the `n` elements are empty handles.

```
public IloNumArray(const IloEnv env, IloInt n, IloNum f0, IloNum f1, ...)
```

This constructor creates an array of `n` floating-point objects for use in a model.

Methods

```
public IloBool contains(IloNum value) const
```

This member function checks whether the value is contained or not.

```
public IloNum & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`.

```
public const IloNum & operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

```
 IloArray operator[] (IloInt i) const;
```

```
public IloNumExprArg operator[](IloIntExprArg intExp) const
```

This subscripting operator returns an expression node for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `intExp` is bound to the value `i`, then the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `intExp`.

This operator is also known as an element expression.

```
public IloIntArray toIntArray() const
```

This member function copies the invoking numeric array to a new instance of `IloIntArray`, checking the type of the values during the copy.

Class IloNumColumn

Definition file: ilconcert/iloexpression.h

For IBM ILOG CPLEX: helps you design a model through column representation.
An instance of this class helps you design a model through column representation. In other words, you can create a model by defining each of its columns as an instance of this class. In particular, an instance of `IloNumColumn` enables you to build a column for a numeric variable (an instance of `IloNumVar`) with information about the extractable objects (such as objectives, constraints, etc.) where that numeric variable may eventually appear, even if the numeric variable has not yet been created.

Usually you populate a column (an instance of this class) with objects returned by the `operator()` of the class (such as `IloObjective::operator()`) where you want to install the newly created variable, as in the examples below.

An instance of `IloNumColumn` keeps a list of those objects returned by `operator()`. In other words, an instance of `IloNumColumn` knows the extractable objects where a numeric variable will be added when it is created.

When you create a new instance of `IloNumVar` with an instance of `IloNumColumn` as an argument, then Concert Technology adds the newly created numeric variable to all the extractable objects (such as constraints, ranges, objectives, etc.) for which an instance of `IloAddNumVar` will be added to this instance of `IloNumColumn`. Note that `IloNumColumn` does not support normalization, as normalization is not well defined for constraints such as `IloSOS1` and `IloAllDiff`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

For information on columnwise modeling, see the concept Column-Wise Modeling.

See Also: IloNumVar, IloObjective, IloRange

Constructor Summary	
public	IloNumColumn(const IloEnv env)
public	IloNumColumn(const IloAddNumVar & var)

Method Summary	
public void	clear() const
public	operator const IloAddNumVar &() const
public IloNumColumn &	operator+=(const IloAddValueToRange & rhs)
public IloNumColumn &	operator+=(const IloAddNumVar & rhs)
public IloNumColumn &	operator+=(const IloNumColumn & rhs)

Constructors

`public` **`IloNumColumn`**`(const IloEnv env)`

This constructor creates an empty column in the environment `env`.

`public` **`IloNumColumn`**`(const IloAddNumVar & var)`

This constructor creates a column and adds `var` to it.

Methods

```
public void clear() const
```

This member function removes (from the invoking column) its list of extractable objects.

```
public operator const IloAddNumVar &() const
```

This casting operator allows you to use instances of `IloNumColumn` in column expressions. It accepts an extractable object, such as an objective (an instance of `IloObjective`) or a constraint (an instance of `IloConstraint`). It returns the object derived from `IloAddNumVar` and needed to represent the extractable object in column format.

```
public IloNumColumn & operator+=(const IloAddValueToRange & rhs)
```

This operator adds the appropriate instances of `IloAddValueToRange` for the righthand side `rhs` to the invoking column.

Examples:

To use an instance of this class to create a column with a coefficient of 2 in the objective, with 10 in `range1`, and with 3 in `range2`, set:

```
IloNumColumn col = obj(2) + range1(10) + range2(3);
```

To use an instance of this class to create a numeric variable corresponding to the column with lower bound 0 (zero) and upper bound 10:

```
IloNumVar var(env, col, 0, 10);
```

Another example:

```
IloNumColumn col1(env);
IloNumColumn col2 = rng7(3.1415);
col1 += obj(1.0);
col1 += rng(-12.0);
col2 += rng2(13.7) + rng3(14.7);
col2 += col1;
```

```
public IloNumColumn & operator+=(const IloAddNumVar & rhs)
```

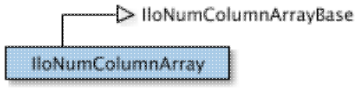
This operator adds the appropriate instances of `IloAddNumVar` for the righthand side `rhs` to the invoking column.

```
public IloNumColumn & operator+=(const IloNumColumn & rhs)
```

This operator assigns the righthand side `rhs` to the invoking column.

Class IloNumColumnArray

Definition file: ilconcert/iloexpression.h



For IBM ILOG CPLEX: array class of the column representation class for a model.
For each basic type, Concert Technology defines a corresponding array class. `IloNumColumnArray` is the array class of the column representation class for a model. The implementation class for `IloNumColumnArray` is the undocumented class `IloNumColumnArrayI`.

Instances of `IloNumColumnArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloModel, IloNumColumn

Constructor Summary	
public	`IloNumColumnArray(IloDefaultArrayI * i=0)`
public	`IloNumColumnArray(const IloEnv env, IloInt n=0)`
public	`IloNumColumnArray(const IloNumColumnArray & h)`

Constructors

`public` **`IloNumColumnArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty extensible array of columns. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

`public` **`IloNumColumnArray`**`(const IloEnv env, IloInt n=0)`

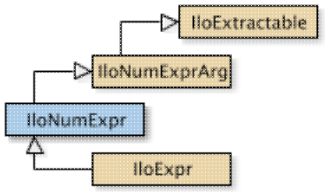
This constructor creates an array of `n` elements. Initially, the `n` elements are empty handles.

`public` **`IloNumColumnArray`**`(const IloNumColumnArray & h)`

This copy constructor creates a handle to the array of column objects specified by `copy`.

Class IloNumExpr

Definition file: ilconcert/iloexpression.h



The class of numeric expressions in a Concert model.
Numeric expressions in Concert Technology are represented using the class `IloNumExpr`.

Constructor Summary	
public	IloNumExpr()
public	IloNumExpr(IloNumExprI * impl)
public	IloNumExpr(const IloNumExprArg expr)
public	IloNumExpr(const IloEnv env, IloNum=0)
public	IloNumExpr(const IloNumLinExprTerm term)
public	IloNumExpr(const IloIntLinExprTerm term)
public	IloNumExpr(const IloExpr & expr)

Method Summary	
public IloNumExprI *	getImpl() const
public IloNumExpr &	operator*=(IloNum val)
public IloNumExpr &	operator+=(const IloNumExprArg expr)
public IloNumExpr &	operator+=(IloNum val)
public IloNumExpr &	operator-=(const IloNumExprArg expr)
public IloNumExpr &	operator-=(IloNum val)
public IloNumExpr &	operator/=(IloNum val)

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Inner Class	
IloNumExpr::NonLinearExpression	The class of exceptions thrown if a numeric constant of a nonlinear expression is set or queried.

Constructors

`public` **`IloNumExpr`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloNumExpr`**`(IloNumExprI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IloNumExpr`**`(const IloNumExprArg expr)`

This constructor creates a numeric expression using the undocumented class `IloNumExprArg`.

`public` **`IloNumExpr`**`(const IloEnv env, IloNum=0)`

This constructor creates a constant numeric expression with the value `n` that the user can modify subsequently with the operators `+=, -=, =` in the environment specified by `env`. It may be used to build other expressions from variables belonging to `env`.

`public` **`IloNumExpr`**`(const IloNumLinExprTerm term)`

This constructor creates a numeric expression using the undocumented class `IloNumLinExprTerm`.

`public` **`IloNumExpr`**`(const IloIntLinExprTerm term)`

This constructor creates a numeric expression using the undocumented class `IloIntLinExprTerm`.

`public` **`IloNumExpr`**`(const IloExpr & expr)`

This is the copy constructor for this class.

Methods

`public IloNumExprI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloNumExpr &` **`operator*=`**`(IloNum val)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x * ...`

`public IloNumExpr &` **`operator+=`**`(const IloNumExprArg expr)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`

```
public IloNumExpr & operator+=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`

```
public IloNumExpr & operator-=(const IloNumExprArg expr)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`

```
public IloNumExpr & operator-=(IloNum val)
```

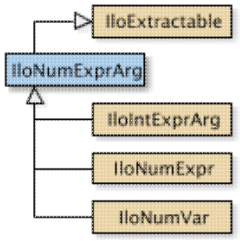
This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`

```
public IloNumExpr & operator/=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x / ...`

Class IloNumExprArg

Definition file: ilconcert/iloexpression.h



A class used internally in Concert Technology.
Concert Technology uses instances of this class internally as temporary objects when it is parsing a C++ expression in order to build an instance of `IloNumExpr`. As a Concert Technology user, you will not need this class yourself; in fact, you should not use them directly. They are documented here because the return value of certain functions, such as `IloSum` or `IloScalProd`, can be an instance of this class.

Constructor Summary	
public	IloNumExprArg()
public	IloNumExprArg(IloNumExprI * impl)

Method Summary	
public IloNumExprI *	getImpl() const

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloNumExprArg**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloNumExprArg**(IloNumExprI * impl)

This constructor creates a handle object from a pointer to an implementation object.

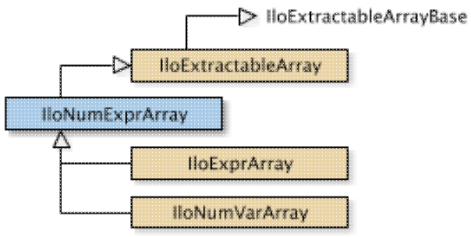
Methods

public IloNumExprI * **getImpl**() const

This member function returns a pointer to the implementation object of the invoking handle.

Class IloNumExprArray

Definition file: ilconcert/iloexpression.h



The array class of the numeric expressions class.
For each basic type, Concert Technology defines a corresponding array class. `IloNumExprArray` is the array class of the numeric expressions class (`IloNumExpr`) for a model.

Instances of `IloNumExprArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Constructor Summary	
public	IloNumExprArray(IloDefaultArrayI * i=0)
public	IloNumExprArray(const IloEnv env, IloInt n=0)

Method Summary	
public void	add(IloInt more, const IloNumExpr x)
public void	add(const IloNumExpr x)
public void	add(const IloNumExprArray array)
public void	endElements()
public IloNumExprArg	operator[](IloIntExprArg anIntegerExpr) const

Inherited Methods from **IloExtractableArray**
add, add, add, endElements, setNames

Constructors

public **IloNumExprArray**(IloDefaultArrayI * i=0)

This constructor creates an empty array of numeric expressions for use in a model. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

public **IloNumExprArray**(const IloEnv env, IloInt n=0)

This constructor creates an array of `n` elements. Initially, the `n` elements are empty handles.

Methods

```
public void add(IloInt more, const IloNumExpr x)
```

This member function appends `x` to the invoking array. The argument `more` specifies how many times.

```
public void add(const IloNumExpr x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloNumExprArray array)
```

This member function appends the elements in `array` to the invoking array.

```
public void endElements()
```

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

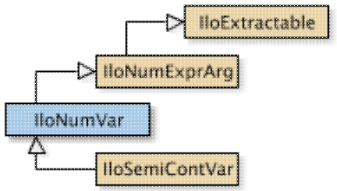
```
public IloNumExprArg operator[](IloIntExprArg anIntegerExpr) const
```

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element expression.

Class IloNumVar

Definition file: ilconcert/iloexpression.h



An instance of this class represents a numeric variable in a model.
An instance of this class represents a numeric variable in a model. A numeric variable may be either an integer variable or a floating-point variable; that is, a numeric variable has a type, a value of the nested enumeration `IloNumVar::Type`. By default, its type is `Float`. It also has a lower and upper bound. A numeric variable cannot assume values less than its lower bound, nor greater than its upper bound.

If you are looking for a class of variables that can assume only constrained integer values, consider the class `IloIntVar`. If you are looking for a class of binary decision variables that can assume only the values 0 (zero) or 1 (one), then consider the class `IloBoolVar`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Programming Hint

For each enumerated value in the nested enumeration `IloNumVar::Type`, Concert Technology offers an equivalent predefined C++ `#define` to make programming easier. For example, in your applications, you may write either statement:

```
IloNumVar x(env, 0, 17, IloNumVar::Int); // using the enumeration

IloNumVar x(env, 0, 17, ILOINT);         // using the#define
```

Note

When numeric bounds are given to an integer variable (an instance of `IloIntVar` or `IloNumVar` with `Type = Int`) in the constructors or via a modifier (`setUB`, `setLB`, `setBounds`), they are inwardly rounded to an integer value. `LB` is rounded down and `UB` is rounded up.

See Also: IloBoolVar, IloIntVar, IloModel, IloNumVarArray, IloNumVar::Type

Constructor Summary	
public	`IloNumVar()`
public	`IloNumVar(IloNumVarI * impl)`
public	`IloNumVar(const IloEnv env, IloNum lb=0, IloNum ub=IloInfinity, IloNumVar::Type type=Float, const char * name=0)`
public	`IloNumVar(const IloEnv env, IloNum lowerBound, IloNum upperBound, const char * name)`
public	`IloNumVar(const IloAddNumVar & var, IloNum lowerBound=0.0, IloNum upperBound=IloInfinity, IloNumVar::Type type=Float, const char * name=0)`
public	`IloNumVar(const IloEnv env, const IloNumArray values, IloNumVar::Type type=Float, const char * name=0)`
public	

	IloNumVar(const IloAddNumVar & var, const IloNumArray values, IloNumVar::Type type=Float, const char * name=0)
public	IloNumVar(const IloConstraint constraint)
public	IloNumVar(const IloNumRange coll, const char * name=0)

Method Summary	
public IloNumVarI *	getImpl() const
public IloNum	getLB() const
public void	getPossibleValues(IloNumArray values) const
public IloNumVar::Type	getType() const
public IloNum	getUB() const
public void	setBounds(IloNum lb, IloNum ub) const
public void	setLB(IloNum num) const
public void	setPossibleValues(const IloNumArray values) const
public void	setUB(IloNum num) const

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Inner Enumeration	
IloNumVar::Type	An enumeration for the class IloNumVar.

Constructors

`public` **`IloNumVar`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloNumVar`**`(IloNumVarI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IloNumVar`**`(const IloEnv env, IloNum lb=0, IloNum ub=IloInfinity, IloNumVar::Type type=Float, const char * name=0)`

This constructor creates a constrained numeric variable and makes it part of the environment `env`. By default, the numeric variable ranges from 0.0 (zero) to the symbolic constant `IloInfinity`, but you can specify other upper and lower bounds yourself. By default, the numeric variable assumes floating-point values. However, you can constrain it to be an integer by setting its `type = Int`. By default, its name is the empty string, but you can specify a `name` of your own choice.

`public` **`IloNumVar`**`(const IloEnv env, IloNum lowerBound, IloNum upperBound, const char * name)`

This constructor creates a constrained numeric variable and makes it part of the environment `env`. The bounds of the variable are set by `lowerBound` and `upperBound`. By default, its name is the empty string, but you can specify a `name` of your own choice.

```
public IloNumVar(const IloAddNumVar & var, IloNum lowerBound=0.0, IloNum
upperBound=IloInfinity, IloNumVar::Type type=Float, const char * name=0)
```

This constructor creates a constrained numeric variable in column format. For more information on adding columns to a model, refer to the concept *Column-Wise Modeling*.

```
public IloNumVar(const IloEnv env, const IloNumArray values, IloNumVar::Type
type=Float, const char * name=0)
```

This constructor creates a constrained discrete numeric variable and makes it part of the environment `env`. The new discrete variable will be limited to values in the set specified by the array `values`. By default, its name is the empty string, but you can specify a name of your own choice. You can use the member function `IloNumVar::setPossibleValues` with instances created by this constructor.

```
public IloNumVar(const IloAddNumVar & var, const IloNumArray values,
IloNumVar::Type type=Float, const char * name=0)
```

This constructor creates a constrained *discrete* numeric variable from `var` by limiting its domain to the values specified in the array `values`. You may use the member function `IloNumVar::setPossibleValues` with instances created by this constructor.

```
public IloNumVar(const IloConstraint constraint)
```

This constructor creates a constrained numeric variable which is equal to the truth value of `constraint`. The truth value of `constraint` is either 0 for `IloFalse` or 1 for `IloTrue`. You can use this constructor to cast a constraint to a constrained numeric variable. That constrained numeric variable can then be used like any other constrained numeric variable. It is thus possible to express sums of constraints, for example. The following line expresses the idea that all three variables cannot be equal:

```
model.add((x != y) + (y != z) + (z != x) >= 2);
```

```
public IloNumVar(const IloNumRange coll, const char * name=0)
```

This constructor creates a constrained *discrete* numeric variable from the given collection

This constructor creates a constrained *discrete* numeric variable from the given collection

Methods

```
public IloNumVarI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getLB() const
```

This member function returns the lower bound of the invoking numeric variable.

```
public void getPossibleValues(IloNumArray values) const
```

This member function accesses the possible values of the invoking numeric variable and puts them in the array `values`.

```
public IloNumVar::Type getType() const
```

This member function returns the type of the invoking numeric variable, specifying whether it is integer (`Int`) or floating-point (`Float`).

```
public IloNum getUB() const
```

This member function returns the upper bound of the invoking numeric variable.

```
public void setBounds(IloNum lb, IloNum ub) const
```

This member function sets `lb` as the lower bound and `ub` as the upper bound of the invoking numeric variable.

Note

The member function `setBounds` notifies Concert Technology algorithms about this change of bounds in this numeric variable.

```
public void setLB(IloNum num) const
```

This member function sets `num` as the lower bound of the invoking numeric variable.

Note

The member function `setLB` notifies Concert Technology algorithms about this change of bound in this numeric variable.

```
public void setPossibleValues(const IloNumArray values) const
```

This member function sets `values` as the domain of the invoking discrete numeric variable. This member function can be called only on instances of `IloNumVar` that have been created with one of the two *discrete* constructors; that is, instances of `IloNumVar` which have been defined by an explicit array of discrete values.

Note

The member function `setPossibleValues` notifies Concert Technology algorithms about this change of domain in this discrete numeric variable.

```
public void setUB(IloNum num) const
```

This member function sets `num` as the upper bound of the invoking numeric variable.

Note

The member function `setUB` notifies Concert Technology algorithms about this change of bound in this numeric variable.

Inner Enumerations

Enumeration Type

Definition file: ilconcert/iloexpression.h

An enumeration for the class IloNumVar.
This nested enumeration enables you to specify whether an instance of `IloNumVar` is of type integer (`Int`), Boolean (`Bool`), or floating-point (`Float`).

Programming Hint

For each enumerated value in `IloNumVar::Type`, there is a predefined equivalent C++ `#define` in the Concert Technology include files to make programming easier. For example, instead of writing `IloNumVar::Int` in your application, you can write `ILOINT`. Likewise, `ILOFLOAT` is defined for `IloNumVar::Float` and `ILOBOOL` for `IloNumVar::Bool`.

See Also: IloNumVar

Fields:

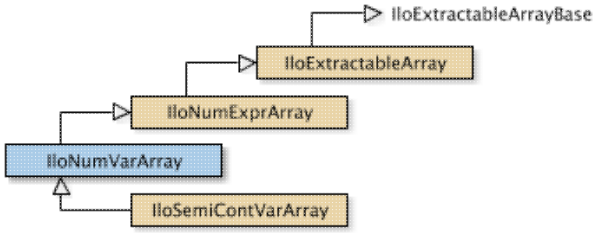
```
Int = 1
```

```
Float = 2
```

```
Bool = 3
```

Class IloNumVarArray

Definition file: ilconcert/iloexpression.h



The array class of IloNumVar.
For each basic type, Concert Technology defines a corresponding array class. `IloNumVarArray` is the array class of the numeric variable class for a model.

Instances of `IloNumVarArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloModel, IloNumVar, operator>>, operator<<

Constructor Summary	
public	`IloNumVarArray(IloDefaultArrayI * i=0)`
public	`IloNumVarArray(const IloEnv env, IloInt n=0)`
public	`IloNumVarArray(const IloEnv env, const IloNumArray lb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)`
public	`IloNumVarArray(const IloEnv env, IloNum lb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)`
public	`IloNumVarArray(const IloEnv env, const IloNumArray lb, IloNum ub, IloNumVar::Type type=ILOFLOAT)`
public	`IloNumVarArray(const IloEnv env, IloInt n, IloNum lb, IloNum ub, IloNumVar::Type type=ILOFLOAT)`
public	`IloNumVarArray(const IloEnv env, const IloNumColumnArray columnarray, IloNumVar::Type type=ILOFLOAT)`
public	`IloNumVarArray(const IloEnv env, const IloNumColumnArray columnarray, const IloNumArray lb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)`

Method Summary	
public void	`add(IloInt more, const IloNumVar x)`
public void	`add(const IloNumVar x)`
public void	`add(const IloNumVarArray array)`
public void	`endElements()`
public IloNumExprArg	`operator[](IloIntExprArg anIntegerExpr) const`
public void	`setBounds(const IloNumArray lb, const IloNumArray ub)`
public IloIntExprArray	`toIntExprArray() const`
public IloIntVarArray	`toIntVarArray() const`

public IloNumExprArray	toNumExprArray() const

Inherited Methods from `IloNumExprArray`
add, add, add, endElements, operator[]

Inherited Methods from `IloExtractableArray`
add, add, add, endElements, setNames

Constructors

```
public IloNumVarArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty extensible array of numeric variables. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) optionally or to use 0 (zero) as a default value of an argument in a constructor.

```
public IloNumVarArray(const IloEnv env, IloInt n=0)
```

This constructor creates an extensible array of `n` numeric variables in `env`. Initially, the `n` elements are empty handles.

```
public IloNumVarArray(const IloEnv env, const IloNumArray lb, const IloNumArray ub,
IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of numeric variables in `env` with lower and upper bounds and type as specified. The instances of `IloNumVar` to fill this array are constructed at the same time. The length of the array `lb` must be the same as the length of the array `ub`. In other words, `lb.getSize == ub.getSize`. This constructor will construct an array with the number of elements in the array `ub`.

```
public IloNumVarArray(const IloEnv env, IloNum lb, const IloNumArray ub,
IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of numeric variables in `env` with lower and upper bounds and type as specified. The instances of `IloNumVar` to fill this array are constructed at the same time. The length of the new array will be the same as the length of the array `ub`.

```
public IloNumVarArray(const IloEnv env, const IloNumArray lb, IloNum ub,
IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of numeric variables in `env` with lower and upper bounds and type as specified. The instances of `IloNumVar` to fill this array are constructed at the same time.

```
public IloNumVarArray(const IloEnv env, IloInt n, IloNum lb, IloNum ub,
IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of `n` numeric variables in `env` with lower and upper bounds and type as specified. The instances of `IloNumVar` to fill this array are constructed at the same time.

```
public IloNumVarArray(const IloEnv env, const IloNumColumnArray columnarray,
IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of numeric variables with type as specified. The instances of `IloNumVar` to fill this array are constructed at the same time.

```
public IloNumVarArray(const IloEnv env, const IloNumColumnArray columnarray, const
IloNumArray lb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of numeric variables with lower and upper bounds and type as specified. The instances of `IloNumVar` to fill this array are constructed at the same time.

Methods

```
public void add(IloInt more, const IloNumVar x)
```

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

```
public void add(const IloNumVar x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloNumVarArray array)
```

This member function appends the elements in `array` to the invoking array.

```
public void endElements()
```

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

```
public IloNumExprArg operator[](IloIntExprArg anIntegerExpr) const
```

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element expression.

```
public void setBounds(const IloNumArray lb, const IloNumArray ub)
```

For each element in the invoking array, this member function sets `lb` as the lower bound and `ub` as the upper bound of the corresponding numeric variable in the invoking array.

Note

The member function `setBounds` notifies Concert Technology algorithms about this change of bounds in this array of numeric variables.

```
public IloIntExprArray toIntExprArray() const
```

This member function copies the invoking array to a new `IloIntExprArray`, checking the type of the variables during the copy.

```
public IloIntVarArray toIntVarArray() const
```

This member function copies the invoking array to a new `IloIntVarArray`, checking the type of the variables during the copy.

```
public IloNumExprArray toNumExprArray() const
```

This member function copies the invoking array to a new `IloNumExprArray`, checking the type of the variables during the copy.

Class IloObjective

Definition file: ilconcert/ilolinear.h

An instance of this class is an objective in a model.
An objective consists of its sense (specifying whether it is a minimization or maximization) and an expression. The expression may be a constant.

An objective belongs to the environment that the variables in its expression belong to. Generally, you will create an objective, add it to a model, and extract the model for an algorithm.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

What Is Extracted

All the variables (that is, instances of `IloNumVar` or one of its subclasses) in the objective (an instance of `IloObjective`) will be extracted when an algorithm such as `IloCplex`, documented in the *IBM ILOG CPLEX Reference Manual*, extracts the objective.

Multiple Objectives

You may create more than one objective in a model, for example, by creating more than one group. However, certain algorithms, such as an instance of `IloCplex`, will throw an exception (on a platform that supports C++ exceptions, when exceptions are enabled) if you attempt to extract more than one objective at a time.

Also see the functions `IloMaximize` and `IloMinimize` for “short cuts” to create objectives.

Normalizing Linear Expressions: Reducing the Terms

Normalizing is sometimes known as *reducing the terms* of a linear expression.

Linear expressions consist of terms made up of constants and variables related by arithmetic operations; for example, x + 3y is a linear expression of two terms consisting of two variables. In some linear expressions, a given variable may appear in more than one term, for example, x + 3y +2x. Concert Technology has more than one way of dealing with linear expressions in this respect, and you control which way Concert Technology treats linear expressions from your application.

In one mode (the default mode), Concert Technology analyzes expressions that your application passes it and attempts to reduce them so that a given variable appears in only one term in the expression. You set this mode with the member function `IloEnv::setNormalizer`.

Certain constructors and member functions in this class check this setting in the model and behave accordingly: they attempt to reduce expressions. This mode may require more time during preliminary computation, but it avoids the possibility of an assertion failing for certain member functions of this class in case of duplicates.

In the other mode, Concert Technology **assumes** that no variable appears in more than one term in any of the linear expressions that your application passes to Concert Technology. We call this mode assume no duplicates. You set this mode with the member function `IloEnv::setNormalizer`.

Certain constructors and member functions in this class check this setting in the model and behave accordingly: they assume that no variable appears in more than one term in an expression. This mode may save time during computation, but it entails the risk that an expression may contain one or more variables, each of which appears in one or more terms. This situation will cause certain `assert` statements in Concert Technology to fail if you do not compile with the flag `-DNDEBUG`.

See Also: IloMaximize, IloMinimize, IloModel, IloObjective::Sense

Constructor Summary	
public	IloObjective()
public	IloObjective(IloObjectiveI * impl)
public	IloObjective(const IloEnv env, IloNum constant=0.0, IloObjective::Sense sense=Minimize, const char * name=0)
public	IloObjective(const IloEnv env, const IloNumExprArg expr, IloObjective::Sense sense=Minimize, const char * name=0)

Method Summary	
public IloNum	getConstant() const
public IloNumExprArg	getExpr() const
public IloObjectiveI *	getImpl() const
public IloObjective::Sense	getSense() const
public void	setConstant(IloNum constant)
public void	setExpr(const IloNumExprArg)
public void	setLinearCoef(const IloNumVar var, IloNum value)
public void	setLinearCoefs(const IloNumVarArray vars, const IloNumArray values)
public void	setSense(IloObjective::Sense sense)

Inherited Methods from **IloExtractable**
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Inner Enumeration	
IloObjective::Sense	Specifies objective as minimization or maximization.

Constructors

public **IloObjective**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloObjective**(IloObjectiveI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IloObjective**(const IloEnv env, IloNum constant=0.0, IloObjective::Sense sense=Minimize, const char * name=0)

This constructor creates an objective consisting of a constant and belonging to env. The sense of the objective (whether it is a minimization or maximization) is specified by sense; by default, it is a minimization. You may supply a name for the objective; by default, its name is the empty string. This constructor is useful when you want to create an empty objective and fill it later by column-wise modeling.

public **IloObjective**(const IloEnv env, const IloNumExprArg expr, IloObjective::Sense

```
sense=Minimize, const char * name=0)
```

This constructor creates an objective to add to a model from `expr`.

After you create an objective from an expression with this constructor, you must use the member function `add` explicitly to add your objective to your model or to a group in order for the objective to be taken into account.

Note

When it accepts an expression as an argument, this constructor checks the setting of IloEnv::setNormalizer to determine whether to assume the expression has already been reduced or to reduce the expression before using it.

Methods

```
public IloNum getConstant() const
```

This member function returns the constant term from the expression of the invoking objective.

```
public IloNumExprArg getExpr() const
```

This member function returns the expression of the invoking `IloObjective` object.

```
public IloObjectiveI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloObjective::Sense getSense() const
```

This member function returns the sense of the invoking objective, specifying whether the objective is a minimization (`Minimize`) or a maximization (`Maximize`).

```
public void setConstant(IloNum constant)
```

This member function sets `constant` as the constant term in the invoking objective, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note

The member function `setConstant` notifies Concert Technology algorithms about this change of this invoking object.

```
public void setExpr(const IloNumExprArg)
```

This member function sets the expression of the invoking `IloObjective` object.

Note

The member function `setExpr` notifies Concert Technology algorithms about this change of this invoking object.

```
public void setLinearCoef(const IloNumVar var, IloNum value)
```

This member function sets `value` as the linear coefficient of the variable `var` in the invoking objective, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note

The member function `setLinearCoef` notifies Concert Technology algorithms about this change of this invoking object.

If you attempt to use `setLinearCoef` on a nonlinear expression, it will throw an exception on platforms that support C++ exceptions when exceptions are enabled.

```
public void setLinearCoefs(const IloNumVarArray vars, const IloNumArray values)
```

For each of the variables in `vars`, this member function sets the corresponding value of `values` (whether integer or floating-point) as its linear coefficient in the invoking objective, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note

The member function `setLinearCoefs` notifies Concert Technology algorithms about this change of this invoking object.

If you attempt to use `setLinearCoef` on a non linear expression, Concert Technology will throw an exception on platforms that support C++ exceptions when exceptions are enabled.

```
public void setSense(IloObjective::Sense sense)
```

This member function sets `sense` to specify whether the invoking objective is a maximization (`Maximize`) or minimization (`Minimize`), and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note

The member function `setSense` notifies Concert Technology algorithms about this change of this invoking object.

Inner Enumerations

Enumeration Sense

Definition file: ilconcert/ilolinear.h

Specifies objective as minimization or maximization.

An instance of the class `IloObjective` represents an objective in a model. This nested enumeration is limited in scope to that class, and its values specify the sense of an objective; that is, whether it is a minimization (`Minimize`) or a maximization (`Maximize`).

See Also: IloObjective

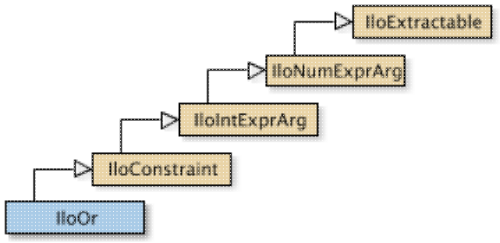
Fields:

`Minimize = 1`

`Maximize = -1`

Class IloOr

Definition file: ilconcert/ilomodel.h



Represents a disjunctive constraint.
An instance of `IloOr` represents a disjunctive constraint. In other words, it defines a disjunctive-OR among any number of constraints. Since an instance of `IloOr` is a constraint itself, you can build up extensive disjunctions by adding constraints to an instance of `IloOr` by means of the member function `IloOr::add`. You can also remove constraints from an instance of `IloOr` by means of the member function `IloOr::remove`.

The elements of a disjunctive constraint must be in the same environment.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Disjunctive Goals

If you would like to represent a disjunctive-OR as a goal (rather than a constraint), then you should consider the function `IloOrGoal`, documented in the *IBM ILOG Solver Reference Manual*.

What Is Extracted

All the constraints (that is, instances of `IloConstraint` or one of its subclasses) that have been added to a disjunctive constraint (an instance of `IloOr`) and that have not been removed from it will be extracted when an algorithm such as `IloCplex`, `IloCP`, or `IloSolver` extracts the constraint.

Example

For example, you may write:

```
IloOr myor(env);
myor.add(constraint1);
myor.add(constraint2);
myor.add(constraint3);
```

Those lines are equivalent to :

```
IloOr or = constraint1 || constraint2 || constraint3;
```

See Also: IloAnd, IloConstraint, operator||

Constructor Summary	
public	IloOr()
public	IloOr(IloOrI * impl)

public	IloOr(const IloEnv env, const char * name=0)

Method Summary	
public void	add(const IloConstraintArray cons) const
public void	add(const IloConstraint con) const
public IloOrI *	getImpl() const
public void	remove(const IloConstraintArray cons) const
public void	remove(const IloConstraint con) const

Inherited Methods from `IloConstraint`
getImpl

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloOr**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloOr**(IloOrI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IloOr**(const IloEnv env, const char * name=0)

This constructor creates a disjunctive constraint for use in `env`. The optional argument `name` is set to `0` by default.

Methods

public void **add**(const IloConstraintArray cons) const

This member function makes all the elements in `array` elements of the invoking disjunctive constraint. In other words, it applies the invoking disjunctive constraint to all the elements of `array`.

Note

The member function `add` notifies Concert Technology algorithms about this change of this invoking object.

```
public void add(const IloConstraint con) const
```

This member function makes `constraint` one of the elements of the invoking disjunctive constraint. In other words, it applies the invoking disjunctive constraint to `constraint`.

Note

The member function `add` notifies Concert Technology algorithms about this change of this invoking object.

```
public IloOrI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void remove(const IloConstraintArray cons) const
```

This member function removes all the elements of `array` from the invoking disjunctive constraint so that the invoking disjunctive constraint no longer applies to any of those elements.

Note

The member function `remove` notifies Concert Technology algorithms about this change of this invoking object.

```
public void remove(const IloConstraint con) const
```

This member function removes `constraint` from the invoking disjunctive constraint so that the invoking disjunctive constraint no longer applies to `constraint`.

Note

The member function `remove` notifies Concert Technology algorithms about this change of this invoking object.

Class IloRandom

Definition file: ilconcert/ilorandom.h

This handle class produces streams of pseudo-random numbers.
You can use objects of this class to create a search with a random element. You can create any number of instances of this class; these instances insure reproducible results in multithreaded applications, where the use of a single source for random numbers creates problems.

See Also the class `IloRandomize` in the *IBM ILOG Solver Reference Manual*.

Constructor Summary	
public	`IloRandom()`
public	`IloRandom(const IloEnv env, IloInt seed=0)`
public	`IloRandom(IloRandomI * impl)`
public	`IloRandom(const IloRandom & rand)`

Method Summary	
public void	`end()`
public IloEnv	`getEnv() const`
public IloNum	`getFloat() const`
public IloRandomI *	`getImpl() const`
public IloInt	`getInt(IloInt n) const`
public const char *	`getName() const`
public IloAny	`getObject() const`
public void	`reSeed(IloInt seed)`
public void	`setName(const char * name) const`
public void	`setObject(IloAny obj) const`

Constructors

`public` **`IloRandom`**`()`

This constructor creates a random number generator; it is initially an empty handle. You must assign this handle before you use its member functions.

`public` **`IloRandom`**`(const IloEnv env, IloInt seed=0)`

This constructor creates an object that generates random numbers. You can seed the generator by supplying a value for the integer argument `seed`.

`public` **`IloRandom`**`(IloRandomI * impl)`

This constructor creates a handle object (an instance of the class `IloRandom`) from a pointer to an implementation object (an instance of the class `IloRandomI`).

```
public IloRandom(const IloRandom & rand)
```

This constructor creates a handle object from a reference to a random number generator. After execution, both the newly constructed handle and `rand` point to the same implementation object.

Methods

```
public void end()
```

This member function releases all memory used by the random number generator. After a call to this member function, you should not use the generator again.

```
public IloEnv getEnv() const
```

This member function returns the environment associated with the implementation class of the invoking generator.

```
public IloNum getFloat() const
```

This member function returns a floating-point number drawn uniformly from the interval `[0..1)`.

```
public IloRandomI * getImpl() const
```

This member function returns the implementation object of the invoking handle.

```
public IloInt getInt(IloInt n) const
```

This member function returns an integer drawn uniformly from the interval `[0..n)`.

```
public const char * getName() const
```

This member function returns a character string specifying the name of the invoking object (if there is one).

```
public IloAny getObject() const
```

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

```
public void reSeed(IloInt seed)
```

This member function re-seeds the random number generator with `seed`.

```
public void setName(const char * name) const
```

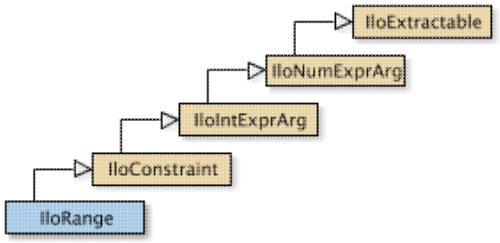
This member function assigns `name` to the invoking object.

```
public void setObject(IloAny obj) const
```

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterward. Normally, `obj` contains user data pertinent to the invoking object.

Class IloRange

Definition file: ilconcert/ilolinear.h



An instance of this class is a range in a model.
This class models a constraint of the form:

```
lowerBound <= expression <= upperBound
```

You can create a range from the constructors in this class or from the arithmetic operators on numeric variables (instances of `IloNumVar` and its subclasses) or on expressions (instances of `IloExpr` and its subclasses):

- `operator <=`
- `operator >=`
- `operator ==`

After you create a constraint, such as an instance of `IloRange`, you must explicitly add it to the model in order for it to be taken into account. To do so, use the member function `IloModel::add` to add the range to a model and the member function `IloAlgorithm::extract` to extract the model for an algorithm.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

What Is Extracted

All the variables (that is, instances of `IloNumVar` or one of its subclasses) in the range (an instance of `IloRange`) will be extracted when an algorithm such as `IloCplex`, documented in the IBM ILOG CPLEX Reference Manual, extracts the range.

Normalizing Linear Expressions: Reducing the Terms

Normalizing is sometimes known as *reducing the terms* of a linear expression.

Linear expressions consist of terms made up of constants and variables related by arithmetic operations; for example, x + 3y. In some linear expressions, a given variable may appear in more than one term, for example, x + 3y +2x. Concert Technology has more than one way of dealing with linear expressions in this respect, and you control which way Concert Technology treats linear expressions from your application.

In one mode (the default mode), Concert Technology analyzes linear expressions that your application passes it, and attempts to reduce them so that a given variable appears in only one term in the expression. You set this mode with the member function `IloEnv::setNormalizer(IloTrue)` .

Certain constructors and member functions in this class check this setting in the model and behave accordingly: they attempt to reduce expressions. This mode may require more time during preliminary computation, but it avoids the possibility of an assertion in some of the member functions of this class failing in the case of duplicates.

In the other mode, Concert Technology assumes that no variable appears in more than one term in any of the linear expressions that your application passes to Concert Technology. We call this mode assume normalized linear expressions. You set this mode with the member function `IloEnv::setNormalizer(IloFalse)`.

Certain constructors and member functions in this class check this setting in the model and behave accordingly: they assume that no variable appears in more than one term in an expression. This mode may save time during computation, but it entails the risk that an expression may contain one or more variables, each of which appears in one or more terms. This situation will cause certain `assert` statements in Concert Technology to fail if you do not compile with the flag `-DNDEBUG`.

See Also: IloConstraint, IloRangeArray

Constructor Summary	
public	IloRange()
public	IloRange(IloRangeI * impl)
public	IloRange(const IloEnv env, IloNum lb, IloNum ub, const char * name=0)
public	IloRange(const IloEnv env, IloNum lhs, const IloNumExprArg expr, IloNum rhs=IloInfinity, const char * name=0)
public	IloRange(const IloEnv env, const IloNumExprArg expr, IloNum rhs=IloInfinity, const char * name=0)

Method Summary	
public IloNumExprArg	getExpr() const
public IloRangeI *	getImpl() const
public IloNum	getLB() const
public IloNum	getUB() const
public IloAddValueToRange	operator()(IloNum value) const
public void	setBounds(IloNum lb, IloNum ub)
public void	setExpr(const IloNumExprArg expr)
public void	setLB(IloNum lb)
public void	setLinearCoef(const IloNumVar var, IloNum value)
public void	setLinearCoefs(const IloNumVarArray vars, const IloNumArray values)
public void	setUB(IloNum ub)

Inherited Methods from `IloConstraint`
getImpl

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

```
public IloRange()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloRange(IloRangeI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloRange(const IloEnv env, IloNum lb, IloNum ub, const char * name=0)
```

This constructor creates an empty range constraint. Before you use this constraint, you must fill the range. The optional argument `name` is set to `0` by default.

After you create a range constraint, you must explicitly add it to a model in order for it to be taken into account. To do so, use the member function `IloModel::add`.

```
public IloRange(const IloEnv env, IloNum lhs, const IloNumExprArg expr, IloNum
rhs=IloInfinity, const char * name=0)
```

This constructor creates a range constraint from an expression (an instance of the class `IloNumExprArg`) and its upper bound (`rhs`). The default bound for `rhs` is the symbolic constant `IloInfinity`. The optional argument `name` is set to `0` by default.

Note

When it accepts an expression as an argument, this constructor checks the setting of `IloEnv::setNormalizer` to determine whether to assume the expression has already been reduced or to reduce the expression before using it.

```
public IloRange(const IloEnv env, const IloNumExprArg expr, IloNum rhs=IloInfinity,
const char * name=0)
```

This constructor creates a range constraint from an expression (an instance of the class `IloNumExprArg`) and its upper bound (`rhs`). Its lower bound (`lhs`) will be `-IloInfinity`. The default bound for `rhs` is `IloInfinity`. The optional argument `name` is set to `0` by default.

Note

When it accepts an expression as an argument, this constructor checks the setting of `IloEnv::setNormalizer` to determine whether to assume the expression has already been reduced or to reduce the expression before using it.

Methods

```
public IloNumExprArg getExpr() const
```

This member function returns the expression of the invoking `IloRange` object.

```
public IloRangeI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getLB() const
```

This member function returns the lower bound of the invoking range.

```
public IloNum getUB() const
```

This member function returns the upper bound of the invoking range.

```
public IloAddValueToRange operator()(IloNum value) const
```

This operator creates the objects needed internally to represent a range in column-wise modeling. See the concept Column-Wise Modeling for an explanation of how to use this operator in column-wise modeling.

```
public void setBounds(IloNum lb, IloNum ub)
```

This member function sets `lb` as the lower bound and `ub` as the upper bound of the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note

The member function `setBounds` notifies Concert Technology algorithms about this change of this invoking object.

```
public void setExpr(const IloNumExprArg expr)
```

This member function sets `expr` as the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note

The member function `setExpr` notifies Concert Technology algorithms about this change of this invoking object.

```
public void setLB(IloNum lb)
```

This member function sets `lb` as the lower bound of the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note

The member function `setLB` notifies Concert Technology algorithms about this change of this invoking object.

```
public void setLinearCoef(const IloNumVar var, IloNum value)
```

This member function sets `value` as the linear coefficient of the variable `var` in the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note

The member function `setLinearCoef` notifies Concert Technology algorithms about this change of this invoking object.

If you attempt to use `setLinearCoef` on a non linear expression, it will throw an exception on platforms that support C++ exceptions when exceptions are enabled.

```
public void setLinearCoefs(const IloNumVarArray vars, const IloNumArray values)
```

For each of the variables in `vars`, this member function sets the corresponding value of `values` (whether integer or floating-point) as its linear coefficient in the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note

The member function `setLinearCoefs` notifies Concert Technology algorithms about this change of this invoking object.

If you attempt to use `setLinearCoef` on a non linear expression, it will throw an exception on platforms that support C++ exceptions when exceptions are enabled.

```
public void setUB(IloNum ub)
```

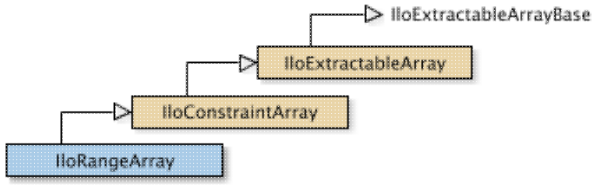
This member function sets `ub` as the upper bound of the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note

The member function `setUB` notifies Concert Technology algorithms about this change of this invoking object.

Class IloRangeArray

Definition file: ilconcert/ilolinear.h



The array class of ranges for a model.
For each basic type, Concert Technology defines a corresponding array class. `IloRangeArray` is the array class of ranges for a model.

Instances of `IloRangeArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

For information on arrays, see the concept Arrays

Note

`IloRangeArray` has access to member functions defined in the `IloArray` template.

See Also: IloRange, operator>>, operator<<

Constructor Summary	
public	`IloRangeArray(IloDefaultArrayI * i=0)`
public	`IloRangeArray(const IloEnv env, IloInt n=0)`
public	`IloRangeArray(const IloEnv env, IloInt n, IloNum lb, IloNum ub)`
public	`IloRangeArray(const IloEnv env, const IloNumArray lbs, const IloNumExprArray rows, const IloNumArray ubs)`
public	`IloRangeArray(const IloEnv env, IloNum lb, const IloNumExprArray rows, const IloNumArray ubs)`
public	`IloRangeArray(const IloEnv env, const IloNumArray lbs, const IloNumExprArray rows, IloNum ub)`
public	`IloRangeArray(const IloEnv env, IloNum lb, const IloNumExprArray rows, IloNum ub)`
public	`IloRangeArray(const IloEnv env, const IloIntArray lbs, const IloNumExprArray rows, const IloIntArray ubs)`
public	`IloRangeArray(const IloEnv env, IloNum lb, const IloNumExprArray rows, const IloIntArray ubs)`
public	`IloRangeArray(const IloEnv env, const IloIntArray lbs, const IloNumExprArray rows, IloNum ub)`
public	`IloRangeArray(const IloEnv env, const IloNumArray lbs, const IloNumArray ubs)`
public	`IloRangeArray(const IloEnv env, const IloIntArray lbs, const IloIntArray ubs)`

public	IloRangeArray(const IloEnv env, IloNum lb, const IloNumArray ubs)
public	IloRangeArray(const IloEnv env, const IloNumArray lbs, IloNum ub)
public	IloRangeArray(const IloEnv env, IloNum lb, const IloIntArray ubs)
public	IloRangeArray(const IloEnv env, const IloIntArray lbs, IloNum ub)

Method Summary	
public void	add(IloInt more, const IloRange range)
public void	add(const IloRange range)
public void	add(const IloRangeArray array)
public IloNumColumn	operator()(const IloNumArray vals)
public IloNumColumn	operator()(const IloIntArray vals)
public IloRange	operator[](IloInt i) const
public IloRange &	operator[](IloInt i)
public void	setBounds(const IloIntArray lbs, const IloIntArray ubs)
public void	setBounds(const IloNumArray lbs, const IloNumArray ubs)

Inherited Methods from **IloConstraintArray**
add, add, add, operator[], operator[]

Inherited Methods from **IloExtractableArray**
add, add, add, endElements, setNames

Constructors

public **IloRangeArray**(IloDefaultArrayI * i=0)

This default constructor creates an empty range array. You cannot create instances of the undocumented class IloDefaultArrayI. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

public **IloRangeArray**(const IloEnv env, IloInt n=0)

This constructor creates an array of n elements, each of which is an empty handle.

public **IloRangeArray**(const IloEnv env, IloInt n, IloNum lb, IloNum ub)

This constructor creates an array of n elements, each with the lower bound lb and the upper bound ub.

public **IloRangeArray**(const IloEnv env, const IloNumArray lbs, const IloNumExprArray rows, const IloNumArray ubs)

This constructor creates an array of ranges from rows, an array of expressions. It uses the corresponding elements of the arrays lbs and ubs to set the lower and upper bounds of elements in the new array. The length of rows must equal the length of lbs and ubs.

```
public IloRangeArray(const IloEnv env, IloNum lb, const IloNumExprArray rows, const
IloNumArray ubs)
```

This constructor creates an array of ranges from `rows`, an array of expressions. The lower bound of every element in the new array will be `lb`. The upper bound of each element of the new array will be the corresponding element of the array `ubs`. The length of `rows` must equal the length of `ubs`.

```
public IloRangeArray(const IloEnv env, const IloNumArray lbs, const IloNumExprArray
rows, IloNum ub)
```

This constructor creates an array of ranges from `rows`, an array of expressions. The upper bound of every element in the new array will be `ub`. The lower bound of each element of the new array will be the corresponding element of the array `lbs`. The length of `rows` must equal the length of `lbs`.

```
public IloRangeArray(const IloEnv env, IloNum lb, const IloNumExprArray rows,
IloNum ub)
```

This constructor creates an array of ranges from `rows`, an array of expressions. The lower bound of every element in the new array will be `lb`. The upper bound of every element in the new array will be `ub`.

```
public IloRangeArray(const IloEnv env, const IloIntArray lbs, const IloNumExprArray
rows, const IloIntArray ubs)
```

This constructor creates an array of ranges from `rows`, an array of expressions. It uses the corresponding elements of the arrays `lbs` and `ubs` to set the lower and upper bounds of elements in the new array. The length of `rows` must equal the length of `lbs` and `ubs`.

```
public IloRangeArray(const IloEnv env, IloNum lb, const IloNumExprArray rows, const
IloIntArray ubs)
```

This constructor creates an array of ranges from `rows`, an array of expressions. The lower bound of every element in the new array will be `lb`. The upper bound of each element of the new array will be the corresponding element of the array `ubs`. The length of `rows` must equal the length of `ubs`.

```
public IloRangeArray(const IloEnv env, const IloIntArray lbs, const IloNumExprArray
rows, IloNum ub)
```

This constructor creates an array of ranges from `rows`, an array of expressions. The upper bound of every element in the new array will be `ub`. The lower bound of each element of the new array will be the corresponding element of the array `lbs`. The length of `rows` must equal the length of `lbs`.

```
public IloRangeArray(const IloEnv env, const IloNumArray lbs, const IloNumArray
ubs)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the arrays `lbs` (or `ubs`). The number of elements in `lbs` must be equal to the number of elements in `ubs`. The lower bound of each element in the new array will be equal to the corresponding element in the array `lbs`. The upper bound of each element in the new array will be equal to the corresponding element in the array

ubs.

```
public IloRangeArray(const IloEnv env, const IloIntArray lbs, const IloIntArray
ubs)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the arrays `lbs` (or `ubs`). The number of elements in `lbs` must be equal to the number of elements in `ubs`. The lower bound of each element in the new array will be equal to the corresponding element in the array `lbs`. The upper bound of each element in the new array will be equal to the corresponding element in the array `ubs`.

```
public IloRangeArray(const IloEnv env, IloNum lb, const IloNumArray ubs)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the array `ubs`. The lower bound of every element in the new array will be equal to `lb`. The upper bound of each element in the new array will be equal to the corresponding element in the array `ubs`.

```
public IloRangeArray(const IloEnv env, const IloNumArray lbs, IloNum ub)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the array `lbs`. The upper bound of every element in the new array will be equal to `ub`. The lower bound of each element in the new array will be equal to the corresponding element in the array `lbs`.

```
public IloRangeArray(const IloEnv env, IloNum lb, const IloIntArray ubs)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the array `ubs`. The lower bound of every element in the new array will be equal to `lb`. The upper bound of each element in the new array will be equal to the corresponding element in the array `ubs`.

```
public IloRangeArray(const IloEnv env, const IloIntArray lbs, IloNum ub)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the array `lbs`. The upper bound of every element in the new array will be equal to `ub`. The lower bound of each element in the new array will be equal to the corresponding element in the array `lbs`.

Methods

```
public void add(IloInt more, const IloRange range)
```

This member function appends `range` to the invoking array multiple times. The argument `more` specifies how many times.

```
public void add(const IloRange range)
```

This member function appends `range` to the invoking array.

```
public void add(const IloRangeArray array)
```

This member function appends the elements in `array` to the invoking array.

```
public IloNumColumn operator()(const IloNumArray vals)
```

This operator constructs ranges in column representation. That is, it creates an instance of `IloNumColumn` that will add a newly created variable to all the ranged constraints in the invoking object, each as a linear term with the corresponding value specified in the array `values`.

```
public IloNumColumn operator()(const IloIntArray vals)
```

This operator constructs ranges in column representation. That is, it creates an instance of `IloNumColumn` that will add a newly created variable to all the ranged constraints in the invoking object, each as a linear term with the corresponding value specified in the array `values`.

```
public IloRange operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

```
 IloRange operator[] (IloInt i) const;
```

```
public IloRange & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`.

```
public void setBounds(const IloIntArray lbs, const IloIntArray ubs)
```

This member function does not change the array itself; instead, it changes the bounds of all the ranged constraints that are elements of the invoking array. At the same time, it also creates an instance of the undocumented class `IloChange` to notify Concert Technology algorithms about this change in an extractable object of the model. The elements of the arrays `lbs` and `ubs` may be integer or floating-point values. The size of the invoking array must be equal to the size of `lbs` and the size of `ubs`.

Note

The member function `setBounds` notifies Concert Technology algorithms about this change of bounds for all the elements in this invoking array.

```
public void setBounds(const IloNumArray lbs, const IloNumArray ubs)
```

This member function does not change the array itself; instead, it changes the bounds of all the ranged constraints that are elements of the invoking array. At the same time, it also creates an instance of the undocumented class `IloChange` to notify Concert Technology algorithms about this change in an extractable object of the model. The elements of the arrays `lbs` and `ubs` may be integer or floating-point values. The size of the invoking array must be equal to the size of `lbs` and the size of `ubs`.

Note

The member function `setBounds` notifies Concert Technology algorithms about this change of bounds for all the elements in this invoking array.

Class IloSemaphore

Definition file: ilconcert/ilothread.h

Provides synchronization primitives.
The class `IloSemaphore` provides synchronization primitives adapted to Concert Technology. This class supports inter-thread communication in multithread applications. An instance of this class, a semaphore, is a counter; its value is always positive. This counter can be incremented or decremented. You can always increment a semaphore, and incrementing is not a blocking operation. However, the value of the counter cannot be negative, so any thread that attempts to decrement a semaphore whose counter is already equal to 0 (zero) is blocked until another thread increments the semaphore.

System Class

`IloSemaphore` is a system class.

Most Concert Technology classes are actually handle classes whose instances point to objects of a corresponding implementation class. For example, instances of the Concert Technology class `IloNumVar` are handles pointing to instances of the implementation class `IloNumVarI`. Their allocation and de-allocation in a Concert Technology environment are managed by an instance of `IloEnv`.

However, system classes, such as `IloSemaphore`, differ from that pattern. `IloSemaphore` is an ordinary C++ class. Its instances are allocated on the C++ heap.

Instances of `IloSemaphore` are not automatically de-allocated by a call to the member function `IloEnv::end`. You must explicitly destroy instances of `IloSemaphore` by means of a call to the delete operator (which calls the appropriate destructor) when your application no longer needs instances of this class.

Furthermore, you should not allocate—neither directly nor indirectly—any instance of `IloSemaphore` in a Concert Technology environment because the destructor for that instance of `IloSemaphore` will never be called automatically by `IloEnv::end` when it cleans up other Concert Technology objects in that Concert Technology environment.

For example, it is not a good idea to make an instance of `IloSemaphore` part of a conventional Concert Technology model allocated in a Concert Technology environment because that instance will not automatically be de-allocated from the Concert Technology environment along with the other Concert Technology objects.

De-allocating Instances of IloSemaphore

Instances of `IloSemaphore` differ from the usual Concert Technology objects because they are not allocated in a Concert Technology environment, and their de-allocation is not managed automatically for you by `IloEnv::end`. Instead, you must explicitly destroy instances of `IloSemaphore` by calling the delete operator when your application no longer needs those objects.

See Also: IloBarrier, IloCondition

Constructor and Destructor Summary	
`public`	`IloSemaphore(int value=0)`
`public`	`~IloSemaphore()`

Method Summary	
`public void`	`post()`
`public int`	`tryWait()`
`public void`	`wait()`

Constructors and Destructors

`public` **`IloSemaphore`**`(int value=0)`

This constructor creates an instance of `IloSemaphore`, initializes it to `value`, and allocates it on the C++ heap (not in a Concert Technology environment). If you do not pass a `value` argument, the constructor initializes the semaphore at 0 (zero).

`public` **`~IloSemaphore`**`()`

The delete operator calls this destructor to de-allocate an instance of `IloSemaphore`. This destructor is called automatically by the runtime system. The destructor de-allocates operating system-specific data structures.

Methods

`public void` **`post`**`()`

This member function increments the invoking semaphore by 1 (one). If there are threads blocked at this semaphore, then this member function wakes one of them.

`public int` **`tryWait`**`()`

This member function attempts to decrement the invoking semaphore by 1 (one). If this decrement leaves the counter positive, then the call succeeds and returns 1 (one). If the decrement would make the counter strictly negative, then the decrement does not occur, the call fails, and the member function returns 0 (zero).

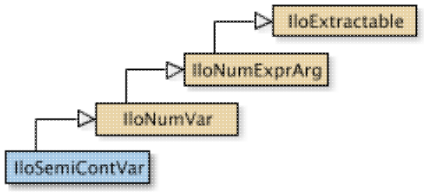
`public void` **`wait`**`()`

This member function decrements the invoking semaphore by 1 (one).

If this decrement would make the semaphore strictly negative, then this member function blocks the calling thread. The thread wakes up when the member function can safely decrement the semaphore without causing the counter to become negative (for example, if another entity increments the semaphore). If this member function cannot decrement the invoking semaphore, then it leads to deadlock.

Class IloSemiContVar

Definition file: ilconcert/iloexpression.h



For IBM ILOG CPLEX: instance represents a constrained semicontinuous variable.
An instance of this class represents a constrained semicontinuous variable in a Concert Technology model. Semicontinuous variables derive from `IloNumVar`, the class of numeric variables.

A semicontinuous variable may be 0 (zero) or it may take a value within an interval defined by its semicontinuous lower and upper bound. Conventionally, semicontinuous variables are defined as floating-point variables, but you can designate an instance of `IloSemiContVar` as integer by using the type specification it inherits from `IloNumVar`. In that case, Concert Technology will impose an integrality constraint on the semicontinuous variable for you, thus further restricting the feasible set of values to 0 (zero) and the integer values in the interval defined by the semicontinuous lower and upper bound.

Note

When numeric bounds are given to an integer variable (an instance of `IloIntVar` or `IloNumVar` with `Type = Int`) in the constructors or via a modifier (`setUB`, `setLB`, `setBounds`), they are inwardly rounded to an integer value. `LB` is rounded down and `UB` is rounded up.

In an instance of `IloNumVar`, `lb` denotes the lower bound of the variable, and `ub` denotes its upper bound. In an instance of the derived class `IloSemiContVar`, `sclb` denotes the semicontinuous lower bound.

In formal terms, if `lb <= 0`, then a semicontinuous variable is a numeric variable with the feasible set of `{0, [sclb, ub]}`, where `0 < sclb < ub`; otherwise, for other values of `lb`, the feasible set of a semicontinuous variable is the intersection of the interval `[lb, ub]` with the set `{0, [sclb, ub]}`. The semicontinuous lower bound `sclb` may differ from the lower bound of an ordinary numeric variable in that the semicontinuous variable is restricted to the semicontinuous region. For example, the table below shows you the bounds of a semicontinuous variable and the corresponding feasible region.

Examples of bounds on semicontinuous variables and their feasible regions

These conditions	define these feasible regions
`lb == ub < sclb`	`{0}` if `lb==ub==0` or empty set if `lb==ub!=0`
`lb <= 0 < sclb < ub`	`{0, [sclb, ub]}`
`0 < lb < sclb < ub`	`[sclb, ub]`
`0 < sclb < lb < ub`	`[lb, ub]`

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloNumVar

Constructor Summary	
`public`	`IloSemiContVar()`

public	IloSemiContVar(IloSemiContVarI * impl)
public	IloSemiContVar(const IloEnv env, IloNum sclb, IloNum ub, IloNumVar::Type type=ILOFLOAT, const char * name=0)
public	IloSemiContVar(const IloAddNumVar & var, IloNum sclb, IloNum ub, IloNumVar::Type type=ILOFLOAT, const char * name=0)

Method Summary	
public IloSemiContVarI *	getImpl() const
public IloNum	getSemiContLB() const
public void	setSemiContLB(IloNum sclb) const

Inherited Methods from IloNumVar
getImpl, getLB, getPossibleValues, getType, getUB, setBounds, setLB, setPossibleValues, setUB

Inherited Methods from IloNumExprArg
getImpl

Inherited Methods from IloExtractable
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloSemiContVar**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloSemiContVar**(IloSemiContVarI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IloSemiContVar**(const IloEnv env, IloNum sclb, IloNum ub, IloNumVar::Type type=ILOFLOAT, const char * name=0)

This constructor creates an instance of IloSemiContVar from its sclb (that is, its semicontinuous lower bound) and its upper bound ub. By default, its type is floating-point, but you can use ILOINT to specify integer; in that case, Concert Technology will impose an integrality constraint on the variable. The value for lb is set to zero.

public **IloSemiContVar**(const IloAddNumVar & var, IloNum sclb, IloNum ub, IloNumVar::Type type=ILOFLOAT, const char * name=0)

This constructor creates an instance of IloSemiContVar from the prototype var.

Methods

public IloSemiContVarI * **getImpl**() const

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getSemiContLB() const
```

This member function returns the semicontinuous lower bound (that is, its `sclb`) of the invoking semicontinuous variable.

```
public void setSemiContLB(IloNum sclb) const
```

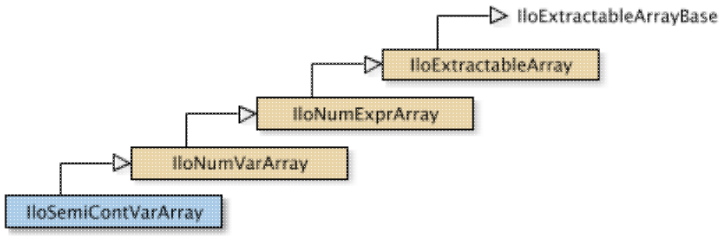
This member function makes `sclb` the semicontinuous lower bound of the invoking semicontinuous variable.

Note

The member function `setSemiContinuousLb` notifies Concert Technology algorithms about this change of this invoking object.

Class IloSemiContVarArray

Definition file: ilconcert/iloexpression.h



For IBM ILOG CPLEX: is the array class of the semicontinuous numeric variable class for a model.
For each basic type, Concert Technology defines a corresponding array class. `IloSemiContVarArray` is the array class of the semicontinuous numeric variable class for a model.

Instances of `IloSemiContVarArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloSemiContVar, IloModel, IloNumVar, operator>>, operator<<

Constructor Summary	
public	`IloSemiContVarArray(IloDefaultArrayI * i=0)`
public	`IloSemiContVarArray(const IloEnv env)`
public	`IloSemiContVarArray(const IloEnv env, IloInt n)`
public	`IloSemiContVarArray(const IloEnv env, IloInt n, IloNum sclb, IloNum ub, IloNumVar::Type type=ILOFLOAT)`
public	`IloSemiContVarArray(const IloEnv env, const IloNumColumnArray columnarray, const IloNumArray sclb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)`

Method Summary	
public void	`add(IloInt more, const IloSemiContVar x)`
public void	`add(const IloSemiContVar x)`
public void	`add(const IloSemiContVarArray array)`

Inherited Methods from `IloNumVarArray`
`add, add, add, endElements, operator[], setBounds, toIntExprArray, toIntVarArray, toNumExprArray`

Inherited Methods from `IloNumExprArray`
`add, add, add, endElements, operator[]`

Inherited Methods from `IloExtractableArray`
`add, add, add, endElements, setNames`

Constructors

```
public IloSemiContVarArray(IloDefaultArrayI * i=0)
```

This constructor creates an empty extensible array of semicontinuous numeric variables. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

```
public IloSemiContVarArray(const IloEnv env)
```

This constructor creates an extensible array of semicontinuous numeric variables in `env`. Initially, the array contains zero elements.

```
public IloSemiContVarArray(const IloEnv env, IloInt n)
```

This constructor creates an extensible array of `n` semicontinuous numeric variables in `env`. Initially, the `n` elements are empty handles.

```
public IloSemiContVarArray(const IloEnv env, IloInt n, IloNum sclb, IloNum ub,
IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of `n` semicontinuous numeric variables in the environment `env`. Each element of the array has a semicontinuous lower bound of `sclb` and an upper bound of `ub`. The type (whether integer, Boolean, or floating-point) of each element is specified by `type`. The default type is floating-point.

```
public IloSemiContVarArray(const IloEnv env, const IloNumColumnArray columnarray,
const IloNumArray sclb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)
```

This constructor creates an extensible array of semicontinuous numeric variables from a column array in the environment `env`. The array `sclb` specifies the corresponding semicontinuous lower bound, and the array `ub` specifies the corresponding upper bound for each new element. The argument `type` specifies the type (whether integer, Boolean, or floating point) of each new element. The default type is floating-point.

Methods

```
public void add(IloInt more, const IloSemiContVar x)
```

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

```
public void add(const IloSemiContVar x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloSemiContVarArray array)
```

This member function appends the elements in `array` to the invoking array.

Class IloSolution

Definition file: ilconcert/ilosolution.h

Instances of this class store solutions to problems.
Instances of this class store solutions to problems. The fundamental property of `IloSolution` is its ability to transfer stored values from or to the active objects associated with it. In particular, the member function `IloSolution::store` stores the values from algorithm variables while the member function `IloSolution::restore` instantiates the actual variables with stored values. Variables in the solution may be selectively restored. This class also offers member functions to copy and to compare solutions.

Information about these classes of variables can be stored in an instance of `IloSolution`:

- `IloAnySet`: the required and possible sets are stored; when the variable is bound, the required and possible sets are equivalent.
- `IloAnyVar`: the value of the variable is stored.
- `IloBoolVar`: the value (true or false) of the variable is stored. Some of the member functions for `IloBoolVar` are covered by the member function for `IloNumVar`, as `IloBoolVar` is a subclass of `IloNumVar`. For example, there is no explicit member function to add objects of type `IloBoolVar`.
- `IloIntSetVar`: the required and possible sets are stored; when the variable is bound, the required and possible sets are equivalent.
- `IloNumVar`: the lower and upper bounds are stored; when the variable is bound, the current lower and upper bound are equivalent.
- `IloIntVar`: the lower and upper bounds are stored; when the variable is bound, the current lower and upper bound are equivalent.
- `IloIntervalVar`: the lower and upper bounds of the start, end, length and size are stored as well as the presence status.
- `IloObjective`: the value of the objective is stored. Objectives are never restored; operations such as `setRestorable` cannot change this. More than one instance of `IloObjective` can be added to a solution,. In such cases, there is the idea of an active objective, which is returned by `IloSolution::getObjective`. The active objective typically specifies the optimization criterion for the problem to which the solution object is a solution. For example, the IBM ILOG Solver class `IloImprove` uses the idea of an active objective.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Objects of type `IloSolution` have a scope, comprising the set of variables that have their values stored in the solution. The scope is given *before* the basic operations of storing and restoring are performed, via `add` and `remove` methods. For example,

```
IloNumVar var(env);
IloSolution soln(env);
solution.add(var);
```

creates a numeric variable and a solution and adds the variable to the solution. Arrays of variables can also be added to the solution. For example,

```
IloNumVarArray arr(env, 10, 0, 1);
soln.add(arr);
```

adds 10 variables with range [0...1]. When an array of variables is added to the solution, the array object itself is not present in the scope of the solution; only the elements are present. If the solution is then stored by means of `soln.store(algorithm)`, the values of variable `var` and `arr[0]` to `arr[9]` are saved. Any attempt to add a variable that is already present in a solution throws an exception, an instance of `IloException`.

Accessors allow access to the stored values of the variables, regardless of the state (or existence) of the algorithm they were stored from. For example,

```
cout << "arr[3] = " << soln.getValue(arr[3]) << endl;
```

Any attempt to access a variable that is not present in the solution throws an instance of `IloException`.

A variable or an array of variables can be removed from a solution. For example,

```
soln.remove(var);
```

removes `var` from the scope of the solution; and

```
soln.remove(arr);
```

removes `arr[0]` to `arr[9]` from the solution.

Any attempt to remove a variable that is not present in the solution throws an instance of `IloException`.

See Also these classes in the *IBM ILOG Solver Reference Manual*: `IloAnySetVar`, `IloAnyVar`, `IloIntSetVar`, `IloStoreSolution`, `IloFRestoreSolution`.

See Also this class in the *IBM ILOG CP Optimizer Reference Manual*: `IloIntervalVar`.

See Also: IloNumVar, IloIntVar, IloObjective

Constructor Summary	
public	IloSolution()
public	IloSolution(IloSolutionI * impl)
public	IloSolution(const IloSolution & solution)
public	IloSolution(IloEnv mem, const char * name=0)

Method Summary	
public void	add(IloAnySetVarArray a) const
public void	add(IloAnySetVar var) const
public void	add(IloAnyVarArray a) const
public void	add(IloAnyVar var) const
public void	add(IloNumVarArray a) const
public void	add(IloNumVar var) const
public void	add(IloObjective objective) const
public IloBool	contains(IloExtractable extr) const
public void	copy(IloExtractable extr, IloSolution solution) const
public void	copy(IloSolution solution) const
public void	end()
public IloEnv	getEnv() const
public IloSolutionI *	getImpl() const
public IloNum	getMax(IloNumVar var) const
public IloNum	getMin(IloNumVar var) const

public const char *	getName() const
public IloAny	getObject() const
public IloObjective	getObjective() const
public IloNum	getObjectiveValue() const
public IloNumVar	getObjectiveVar() const
public IloAnySet	getPossibleSet(IloAnySetVar var) const
public IloAnySet	getRequiredSet(IloAnySetVar var) const
public IloAny	getValue(IloAnyVar var) const
public IloNum	getValue(IloNumVar var) const
public IloNum	getValue(IloObjective obj) const
public IloBool	isBetterThan(IloSolution solution) const
public IloBool	isBound(IloAnySetVar var) const
public IloBool	isBound(IloNumVar var) const
public IloBool	isEquivalent(IloExtractable extr, IloSolution solution) const
public IloBool	isEquivalent(IloSolution solution) const
public IloBool	isObjectiveSet() const
public IloBool	isRestorable(IloExtractable extr) const
public IloBool	isWorseThan(IloSolution solution) const
public IloSolution	makeClone(IloEnv env) const
public void	operator=(const IloSolution & solution)
public void	remove(IloExtractableArray extr) const
public void	remove(IloExtractable extr) const
public void	restore(IloExtractable extr, IloAlgorithm algorithm) const
public void	restore(IloAlgorithm algorithm) const
public void	setFalse(IloBoolVar var) const
public void	setName(const char * name) const
public void	setNonRestorable(IloExtractableArray array) const
public void	setNonRestorable(IloExtractable extr) const
public void	setObject(IloAny obj) const
public void	setObjective(IloObjective objective) const
public void	setPossibleSet(IloAnySetVar var, IloAnySet possible) const
public void	setRequiredSet(IloAnySetVar var, IloAnySet required) const
public void	setRestorable(IloExtractableArray array) const
public void	setRestorable(IloExtractable ex) const
public void	setTrue(IloBoolVar var) const
public void	setValue(IloAnyVar var, IloAny value) const
public void	setValue(IloObjective objective, IloNum value) const
public void	store(IloExtractable extr, IloAlgorithm algorithm) const
public void	store(IloAlgorithm algorithm) const
public void	unsetObjective() const

Inner Class	
IloSolution::Iterator	It allows you to traverse the variables in a solution.

Constructors

`public` **`IloSolution`**`()`

This constructor creates a solution whose implementation pointer is 0 (zero). The handle must be assigned before its methods can be used.

`public` **`IloSolution`**`(IloSolutionI * impl)`

This constructor creates a handle object (an instance of the class `IloSolution`) from a pointer to an implementation object (an instance of the class `IloSolutionI`).

`public` **`IloSolution`**`(const IloSolution & solution)`

This constructor creates a handle object from a reference to a solution. After execution, both the newly constructed handle and `solution` point to the same implementation object.

`public` **`IloSolution`**`(IloEnv mem, const char * name=0)`

This constructor creates an instance of the `IloSolution` class. The optional argument `name`, if supplied, becomes the name of the created object.

Methods

`public void` **`add`**`(IloAnySetVarArray a) const`

This member function adds each element of `array` to the invoking solution.

`public void` **`add`**`(IloAnySetVar var) const`

This member function adds the set variable `var` to the invoking solution.

`public void` **`add`**`(IloAnyVarArray a) const`

This member function adds each element of `array` to the invoking solution.

`public void` **`add`**`(IloAnyVar var) const`

This member function adds the variable `var` to the invoking solution.

`public void` **`add`**`(IloNumVarArray a) const`

This member function adds each element of `array` to the invoking solution.

```
public void add(IloNumVar var) const
```

This member function adds the variable `var` to the invoking solution.

```
public void add(IloObjective objective) const
```

This member function adds `objective` to the invoking solution. If the solution has no active objective, then `objective` becomes the active objective. Otherwise, the active objective remains unchanged.

```
public IloBool contains(IloExtractable extr) const
```

This member function returns `IloTrue` if `extr` is present in the invoking object. Otherwise, it returns `IloFalse`.

```
public void copy(IloExtractable extr, IloSolution solution) const
```

This member function copies the saved value of `extr` from `solution` to the invoking solution. If `extr` does not exist in either `solution` or the invoking object, this member function throws an instance of `IloException`. The restorable status of `extr` is not copied.

```
public void copy(IloSolution solution) const
```

For each variable that has been added to `solution`, this member function copies its saved data to the invoking solution. If a particular extractable does not already exist in the invoking solution, it is automatically added first. If variables were added to the invoking solution, their restorable status is the same as in `solution`. Otherwise, their status remains unchanged in the invoking solution.

```
public void end()
```

This member function deallocates the memory used to store the solution. If you no longer need a solution, calling this member function can reduce memory consumption.

```
public IloEnv getEnv() const
```

This member function returns the environment specified when the invoking object was constructed.

```
public IloSolutionI * getImpl() const
```

This member function returns a pointer to the implementation object corresponding to the invoking solution.

```
public IloNum getMax(IloNumVar var) const
```

This member function returns the maximal value of the variable `var` in the invoking solution.

```
public IloNum getMin(IloNumVar var) const
```

This member function returns the minimal value of the variable `var` in the invoking solution.

```
public const char * getName() const
```

This member function returns a character string specifying the name of the invoking object (if there is one).

```
public IloAny getObject() const
```

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

```
public IloObjective getObjective() const
```

This member function returns the *active* objective as set via a previous call to `IloSolution::add` or `setObjective(IloObjective)`. If there is no active objective, an empty handle is returned.

```
public IloNum getObjectiveValue() const
```

This member function returns the saved value of the current active objective. It can be seen as performing the action `getValue(getObjective())`.

```
public IloNumVar getObjectiveVar() const
```

If the active objective corresponds to a simple `IloNumVar`, this member function returns that variable. If there is no active objective or if the objective is not a simple variable, an empty handle is returned.

```
public IloAnySet getPossibleSet(IloAnySetVar var) const
```

This member function returns the set of possible values for the variable `var`, as stored in the invoking solution.

```
public IloAnySet getRequiredSet(IloAnySetVar var) const
```

This member function returns the set of required values for the variable `var`, as stored in the invoking solution.

```
public IloAny getValue(IloAnyVar var) const
```

This member function returns the value of the variable `var` in the invoking solution.

```
public IloNum getValue(IloNumVar var) const
```

This member function returns the value of the variable `var` in the invoking solution. If the saved minimum and maximum of the variable are not equal, this member function throws an instance of `IloException`.

```
public IloNum getValue(IloObjective obj) const
```

This member function returns the saved value of objective `objective` in the invoking solution.

```
public IloBool isBetterThan(IloSolution solution) const
```

This member function returns `IloTrue` if the invoking solution and `solution` have the same objective and if the invoking solution has a strictly higher quality objective value (according to the sense of the objective). In all other situations, it returns `IloFalse`.

```
public IloBool isBound(IloAnySetVar var) const
```

This member function returns `IloTrue` if the stored required and possible sets for the set variable `var` are equal in the invoking solution. Otherwise, it returns `IloFalse`.

```
public IloBool isBound(IloNumVar var) const
```

This member function returns `IloTrue` if `var` takes a single value in the invoking solution. Otherwise, it returns `IloFalse`.

```
public IloBool isEquivalent(IloExtractable extr, IloSolution solution) const
```

This member function returns `IloTrue` if the saved value of `extr` is the same in the invoking solution and `solution`. Otherwise, it returns `IloFalse`. If `extr` does not exist in either `solution` or the invoking object, the member function throws an instance of `IloException`.

```
public IloBool isEquivalent(IloSolution solution) const
```

This member function returns `IloTrue` if the invoking object and `solution` contain the same variables with the same saved values. Otherwise, it returns `IloFalse`.

```
public IloBool isObjectiveSet() const
```

This member function returns `IloTrue` if the invoking solution has an active objective. Otherwise, it returns `IloFalse`.

```
public IloBool isRestorable(IloExtractable extr) const
```

This member function returns `IloFalse` if `setNonRestorable(extr)` was called more recently than `setRestorable(extr)`. Otherwise, it returns `IloTrue`. This member function always returns `IloFalse` when it is passed an `IloObjective` object.

```
public IloBool isWorseThan(IloSolution solution) const
```

This member function returns `IloTrue` if the invoking solution and `solution` have the same objective and if the invoking solution has a strictly lower quality objective value (according to the sense of the objective). In all other situations, it returns `IloFalse`.

```
public IloSolution makeClone(IloEnv env) const
```

This member function allocates a new solution on `env` and adds to it all variables that were added to the invoking object. The “restorable” status of all variables in the clone is the same as that in the invoking solution. Likewise, the active objective in the clone is the same as that in the invoking solution. The newly created solution is returned.

```
public void operator=(const IloSolution & solution)
```

This operator assigns an address to the handle pointer of the invoking solution. That address is the location of the implementation object of `solution`. After the execution of this operator, the invoking solution and `solution` both point to the same implementation object.

```
public void remove(IloExtractableArray extr) const
```

This member function removes each element of `array` from the invoking solution. If the invoking solution does not contain all elements of `array`, the member function throws an instance of `IloException`.

```
public void remove(IloExtractable extr) const
```

This member function removes extractable `extr` from the invoking solution. If the invoking solution does not contain `extr`, the member function throws an instance of `IloException`.

```
public void restore(IloExtractable extr, IloAlgorithm algorithm) const
```

This member function restores the value of the extractable corresponding to `extr` by reference to `algorithm`. The use of this member function depends on the state of `algorithm`. If `algorithm` is an instance of the IBM ILOG Solver class `IloSolver`, this member function can only be used during search. If `extr` does not exist in the invoking solution, the member function throws an instance of `IloException`.

```
public void restore(IloAlgorithm algorithm) const
```

This member function uses `algorithm` to instantiate the variables in the invoking solution with their saved values. The value of any objective added to the solution is not restored. The use of this member function depends on the state of `algorithm`. If `algorithm` is an instance of the IBM ILOG Solver class `IloSolver`, this member function can only be used during search.

```
public void setFalse(IloBoolVar var) const
```

This member function sets the stored value of `var` to `IloFalse` in the invoking solution.

```
public void setName(const char * name) const
```

This member function assigns `name` to the invoking object.

```
public void setNonRestorable(IloExtractableArray array) const
```

This member function specifies to the invoking solution that when the solution is restored by means of `restore(IloAlgorithm)` or `restore(IloExtractable, IloAlgorithm)`, no elements of `array` will be restored. When an array of variables is added to a solution, each variable is added in a “restorable” state.

```
public void setNonRestorable(IloExtractable extr) const
```

This member function specifies to the invoking solution that when the solution is restored by means of `restore(IloAlgorithm)` or `restore(IloExtractable, IloAlgorithm)`, `extr` will not be restored. When a variable is added to a solution, it is added in a “restorable” state.

```
public void setObject(IloAny obj) const
```

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterward. Normally, `obj` contains user data pertinent to the invoking object.

```
public void setObjective(IloObjective objective) const
```

This member function adds `objective` to the invoking solution, if it is not already present, and sets the active objective to `objective`.

```
public void setPossibleSet(IloAnySetVar var, IloAnySet possible) const
```

This member function sets the stored possible values for `var` as `possible` in the invoking solution.

```
public void setRequiredSet(IloAnySetVar var, IloAnySet required) const
```

This member function sets the stored required values for `var` as `required` in the invoking solution.

```
public void setRestorable(IloExtractableArray array) const
```

This member function specifies to the invoking solution that when the solution is restored by means of `restore(IloAlgorithm)` or `restore(IloExtractable, IloAlgorithm)`, the appropriate element(s) of `array` will be restored. When an array of variables is added to a solution, each variable is added in a “restorable” state. This call has no effect on objects of type `IloObjective`; objects of this type are never restored.

```
public void setRestorable(IloExtractable ex) const
```

This member function specifies to the invoking solution that when the solution is restored by means of `restore(IloAlgorithm)` or `restore(IloExtractable, IloAlgorithm)`, `extr` will be restored. When a variable is added to a solution, it is added in a “restorable” state. This call has no effect on objects of type `IloObjective`; objects of that type are never restored.

```
public void setTrue(IloBoolVar var) const
```

This member function sets the stored value of `var` to `IloTrue` in the invoking solution.

```
public void setValue(IloAnyVar var, IloAny value) const
```

This member function sets the value of the variable `var` to `value` in the invoking solution.

```
public void setValue(IloObjective objective, IloNum value) const
```

This member function sets the value of `objective` as stored in the invoking solution to `value`. This member function should be used with care and only when the objective value of the solution is known exactly.

```
public void store(IloExtractable extr, IloAlgorithm algorithm) const
```

This member function stores the value of the extractable corresponding to `extr` by reference to `algorithm`. If `extr` does not exist in the invoking solution, the member function throws an instance of `IloException`.

```
public void store(IloAlgorithm algorithm) const
```

This member function stores the values of the objects added to the solution by reference to `algorithm`.

```
public void unsetObjective() const
```

This member function asserts that there should be no active objective in the invoking solution, although the previous active object is still present. A new active objective can be set via `IloSolution::add` or `IloSolution::setObjective`.

Class IloSolutionIterator<>

Definition file: ilconcert/ilosolution.h

This template class creates a typed iterator over solutions.
You can use this iterator to discover all extractable objects added to a solution and of a particular type. The type is denoted by `E` in the template.

This iterator is not robust. If the variable at the current position is deleted from the solution being iterated over, the behavior of this iterator afterward is undefined.

An iterator created with this template differs from an instance of `IloSolution::Iterator`. An instance of `IloSolution::Iterator` works on all extractable objects within a given solution (an instance of `IloSolution`). In contrast, an iterator created with this template only iterates over extractable objects of the specified type.

See Also: IloSolution, IloSolution::Iterator

Constructor Summary	
public	IloSolutionIterator(IloSolution s)

Method Summary	
public E	operator*() const
public void	operator++()

Constructors

`public` **`IloSolutionIterator`**`(IloSolution s)`

This constructor creates an iterator for instances of the class `E`.

Methods

`public E` **`operator*`**`() const`

This operator returns the current element, the one to which the invoking iterator points. This current element is a handle to an extractable object (not a pointer to the implementation object).

`public void` **`operator++`**`()`

This operator advances the iterator by one position.

Class IloSolutionManip

Definition file: ilconcert/ilosolution.h

An instance of this class accesses a specific part of a solution.
To display a specific part of the solution, you construct the class `IloSolutionManip` from a solution and an extractable object. You use the `operator<<` with this constructed class to display information stored on the specified extractable object in the solution.

See Also: IloSolution, operator<<

Constructor Summary	
`public`	`IloSolutionManip(IloSolution solution, IloExtractable extr)`

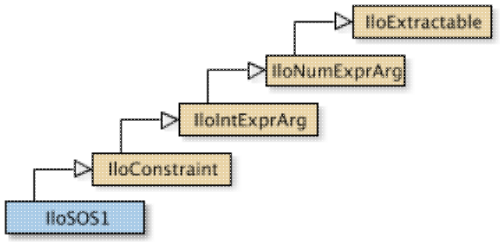
Constructors

`public` **`IloSolutionManip`**`(IloSolution solution, IloExtractable extr)`

This constructor creates an instance of `IloSolutionManip` from the solution specified by `solution` and from the extractable object `extr`. The constructor throws an exception (an instance of `IloException`) if `extr` has not been added to `solution`. You can use the `operator<<` with the newly created object to display the information in `extr` stored in `solution`.

Class IloSOS1

Definition file: ilconcert/ilolinear.h



For IBM ILOG CPLEX: represents special ordered sets of type 1 (SOS1).
This handle class represents special ordered sets of type 1 (SOS1). A special ordered set of type 1 specifies a set of variables, and only one among them may take a non zero value. You may assign a weight to each variable in an SOS1. This weight specifies an order among the variables. If you do not assign any weights to enforce order among the variables, then Concert Technology considers the order in which you gave the variables to the constructor of this set and the order in which you added variables later.

When you extract a model (an instance of `IloModel`) for an instance of `IloCplex` (documented in the *IBM ILOG CPLEX Reference Manual*), it will use the order for branching on variables.

For more details about SOS1, see the *IBM ILOG CPLEX Reference* and *User Manuals*.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloSOS1Array, IloSOS2

Constructor Summary	
public	`IloSOS1()`
public	`IloSOS1(IloSOS1I * impl)`
public	`IloSOS1(const IloEnv env, const char * name=0)`
public	`IloSOS1(const IloEnv env, const IloNumVarArray vars, const char * name=0)`
public	`IloSOS1(const IloEnv env, const IloNumVarArray vars, const IloNumArray vals, const char * name=0)`

Method Summary	
public IloSOS1I *	`getImpl() const`
public void	`getNumVars(IloNumVarArray variables) const`
public void	`getValues(IloNumArray values) const`

Inherited Methods from `IloConstraint`
`getImpl`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

```
public IloSOS1()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloSOS1(IloSOS1I * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloSOS1(const IloEnv env, const char * name=0)
```

This constructor creates a special ordered set of type 1 (SOS1). You must add the variables to this set for them to be taken into account.

```
public IloSOS1(const IloEnv env, const IloNumVarArray vars, const char * name=0)
```

This constructor creates a special ordered set of type 1 (SOS1). The set includes each of the variables specified in the array `vars`.

```
public IloSOS1(const IloEnv env, const IloNumVarArray vars, const IloNumArray vals,
const char * name=0)
```

This constructor creates a special ordered set of type 1 (SOS1). The set includes the variables specified in the array `vars`. The corresponding value in `vals` specifies the weight of each variable in `vars`.

Methods

```
public IloSOS1I * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void getNumVars(IloNumVarArray variables) const
```

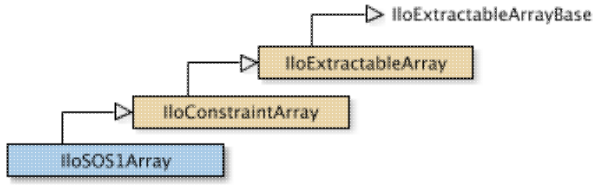
This member function accesses the variables in a special ordered set of type 1 (SOS1) and puts those variables into its argument `variables`.

```
public void getValues(IloNumArray values) const
```

This member function accesses the weights of the variables in a special ordered set of type 1 (SOS1) and puts those weights into its argument `values`.

Class IloSOS1Array

Definition file: ilconcert/ilolinear.h



For IBM ILOG CPLEX: the array class of special ordered sets of type 1 (SOS1).
For each basic type, Concert Technology defines a corresponding array class. `IloSOS1Array` is the array class of special ordered sets of type 1 (SOS1) for a model.

Instances of `IloSOS1Array` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

For information on arrays, see the concept Arrays

See Also: IloSOS1, operator>>

Constructor Summary	
public	IloSOS1Array(IloDefaultArrayI * i=0)
public	IloSOS1Array(const IloEnv env, IloInt n=0)

Method Summary	
public void	add(IloInt more, const IloSOS1 & x)
public void	add(const IloSOS1 & x)
public void	add(const IloSOS1Array & x)
public IloSOS1	operator[](IloInt i) const
public IloSOS1 &	operator[](IloInt i)

Inherited Methods from `IloConstraintArray`
add, add, add, operator[], operator[]

Inherited Methods from `IloExtractableArray`
add, add, add, endElements, setNames

Constructors

`public` **`IloSOS1Array`**`(IloDefaultArrayI * i=0)`

This default constructor creates an empty array. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

```
public IloSOS1Array(const IloEnv env, IloInt n=0)
```

This constructor creates an array of `n` empty elements in the environment `env`.

Methods

```
public void add(IloInt more, const IloSOS1 & x)
```

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

```
public void add(const IloSOS1 & x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloSOS1Array & x)
```

This member function appends the elements in `array` to the invoking array.

```
public IloSOS1 operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

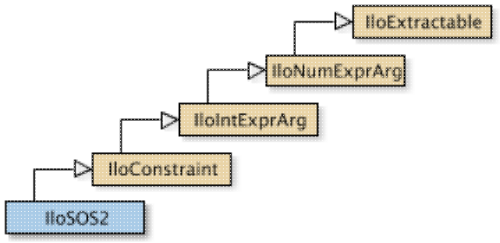
```
IloSOS1 operator[] (IloInt i) const;
```

```
public IloSOS1 & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`.

Class IloSOS2

Definition file: ilconcert/ilolinear.h



For IBM ILOG CPLEX: represents special ordered sets of type 2 (SOS2).
This handle class represents special ordered sets of type 2 (SOS2). A special ordered set of type 2 specifies a set of variables, and only two among them may take a non zero value. These two variables must be adjacent. You may assign a weight to each variable in an SOS2. This weight specifies an order among the variables. Concert Technology asserts adjacency with respect to this assigned order. If you do not assign any weights to enforce order and adjacency among the variables, then Concert Technology considers the order in which you gave the variables to the constructor of this set and the order in which you added variables later (for example, by column generation).

When you extract a model (an instance of `IloModel`) for an instance of `IloCplex` (documented in the *IBM ILOG CPLEX Reference Manual*), it will use the order of the SOS2 for branching on variables.

For more details about SOS2, see the *IBM ILOG CPLEX Reference* and *User's Manuals*. Special ordered sets of type 2 (SOS2) commonly appear in models of piecewise linear functions. Concert Technology provides direct support for piecewise linear models in `IloPiecewiseLinear`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloPiecewiseLinear, IloSOS1, IloSOS2Array

Constructor Summary	
public	`IloSOS2()`
public	`IloSOS2(IloSOS2I * impl)`
public	`IloSOS2(const IloEnv env, const char * name=0)`
public	`IloSOS2(const IloEnv env, const IloNumVarArray vars, const char * name=0)`
public	`IloSOS2(const IloEnv env, const IloNumVarArray vars, const IloNumArray vals, const char * name=0)`

Method Summary	
public IloSOS2I *	`getImpl() const`
public void	`getNumVars(IloNumVarArray variables) const`
public void	`getValues(IloNumArray values) const`

Inherited Methods from `IloConstraint`
`getImpl`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

```
public IloSOS2()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloSOS2(IloSOS2I * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IloSOS2(const IloEnv env, const char * name=0)
```

This constructor creates a special ordered set of type 2 (SOS2). You must add the variables to this set for them to be taken into account.

```
public IloSOS2(const IloEnv env, const IloNumVarArray vars, const char * name=0)
```

This constructor creates a special ordered set of type 2 (SOS2). The set includes each of the variables specified in the array `vars`.

```
public IloSOS2(const IloEnv env, const IloNumVarArray vars, const IloNumArray vals,
const char * name=0)
```

This constructor creates a special ordered set of type 2 (SOS2). The set includes the variables specified in the array `vars`. The corresponding value in `vals` specifies the weight of each variable in `vars`.

Methods

```
public IloSOS2I * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void getNumVars(IloNumVarArray variables) const
```

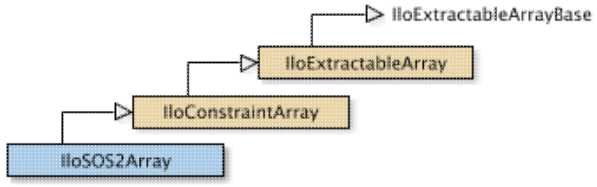
This member function accesses the variables in a special ordered set of type 2 (SOS2) and puts those variables into its argument `variables`.

```
public void getValues(IloNumArray values) const
```

This member function accesses the weights of the variables in a special ordered set of type 2 (SOS2) and puts those weights into its argument `values`.

Class IloSOS2Array

Definition file: ilconcert/ilolinear.h



For IBM ILOG CPLEX: the array class of special ordered sets of type 2 (SOS2).
For each basic type, Concert Technology defines a corresponding array class. `IloSOS2Array` is the array class of special ordered sets of type 2 (SOS2) for a model.

Instances of `IloSOS2Array` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

For information on arrays, see the concept Arrays

See Also: IloSOS2, operator>>

Constructor Summary	
public	IloSOS2Array(IloDefaultArrayI * i=0)
public	IloSOS2Array(const IloEnv env, IloInt num=0)

Method Summary	
public void	add(IloInt more, const IloSOS2 x)
public void	add(const IloSOS2 x)
public void	add(const IloSOS2Array array)
public IloSOS2	operator[](IloInt i) const
public IloSOS2 &	operator[](IloInt i)

Inherited Methods from `IloConstraintArray`
add, add, add, operator[], operator[]

Inherited Methods from `IloExtractableArray`
add, add, add, endElements, setNames

Constructors

`public` **`IloSOS2Array`**`(IloDefaultArrayI * i=0)`

This default constructor creates an empty array. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

```
public IloSOS2Array(const IloEnv env, IloInt num=0)
```

This constructor creates an array of `num` empty elements in the environment `env`.

Methods

```
public void add(IloInt more, const IloSOS2 x)
```

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

```
public void add(const IloSOS2 x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloSOS2Array array)
```

This member function appends the elements in `array` to the invoking array.

```
public IloSOS2 operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

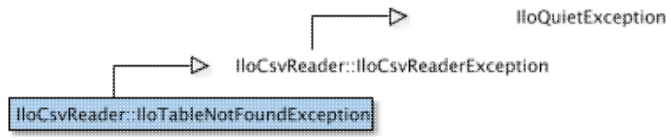
```
 IloSOS2 operator[] (IloInt i) const;
```

```
public IloSOS2 & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`.

Class IloCsvReader::IloTableNotFoundException

Definition file: ilconcert/ilocsvreader.h



Exception thrown for unfound table.
This exception is thrown by the constructor `IloCsvTableReader(IloCsvReaderI *, const char * name = 0)` and by the member functions listed below if the table you want to construct or to get is not found.

- IloCsvReader::getTableByNumber
- IloCsvReader::getTableByName
- IloCsvReader::getTable

Class IloTimer

Definition file: ilconcert/iloenv.h

Represents a timer.
An instance of `IloTimer` represents a timer in a Concert Technology model. It works like a stop watch. The timer report the CPU time. On multi threaded environment, we summed the CPU time used by each thread.

See Also: IloEnv

Constructor Summary	
public	IloTimer(const IloEnv env)

Method Summary	
public IloEnv	getEnv() const
public IloNum	getTime() const
public void	reset()
public IloNum	restart()
public IloNum	start()
public IloNum	stop()

Constructors

public **IloTimer**(const IloEnv env)

This constructor creates a timer.

Methods

public IloEnv **getEnv**() const

This constructor creates an instance of the class `IloTimer`

This member function returns the environment in which the invoking timer was constructed.

public IloNum **getTime**() const

This member function returns the accumulated time, in seconds, since one of these conditions:

- the first call of the member function `start` after construction of the invoking timer;
- the most recent call to the member function `restart`;
- a call to `reset`.

public void **reset**()

This member function sets the elapsed time of the invoking timer to 0.0. It also stops the clock.

`public IloNum` **`restart`**`()`

This member function returns the accumulated time, resets the invoking timer to 0.0, and starts the timer again. In other words, the member function `restart` is equivalent to the member function `reset` followed by `start`.

`public IloNum` **`start`**`()`

This member function makes the invoking timer resume accumulating time. It returns the time accumulated so far.

`public IloNum` **`stop`**`()`

This member function stops the invoking timer so that it no longer accumulates time.

Class IloXmlContext

Definition file: ilconcert/iloxmlcontext.h

An instance of `IloXmlContext` allows you to serialize an `IloModel` or an `IloSolution` in XML.

You can write an `IloModel` using `IloXmlContext::writeModel`, write an `IloSolution` using `IloXmlContext::writeSolution`, or write both using `IloXmlContext::writeModelAndSolution`.

You can read an `IloModel` in XML using `IloXmlContext::readModel`, read an `IloSolution` in XML using `IloXmlContext::readSolution`, or read both using `IloXmlContext::readModelAndSolution`.

Other products should add their own serialization class and add them to the plug-in using the member functions `IloXmlContext::registerXML` and `IloXmlContext::registerXMLArray`.

Examples

For example, you can write:

```
IloModel model(env);
      IloSolution solution(env);
 ...;
IloXmlContext context(env);
context.writeModel(model, "model.xml");
context.writeSolution(solution, "solution.xml");
```

or you can write

```
IloModel model(env);
      IloSolution solution(env);
      IloXmlContext context(env);
      context.readModel(model, "model.xml");
      context.readSolution(solution, "solution.xml");
```

See Also: IloXmlReader, IloXmlWriter, IloXmlInfo

Constructor Summary	
public	IloXmlContext(IloEnv env, const char * name=0)
public	IloXmlContext(IloXmlContextI * impl=0)

Method Summary	
public void	end()
public IloInt	getChildIdReadError() const
public const char *	getChildTagReadError() const
public IloIntArray	getIdListReadError() const
public IloXmlContextI *	getImpl() const
public IloInt	getParentIdReadError() const
public const char *	getParentTagReadError() const
public IloAnyArray	getTagListReadError() const
public const char *	getWriteError() const

public int	getWritePrecision() const
public IloBool	readModel(IloModel model, istream & file) const
public IloBool	readModel(IloModel model, const char * fileName) const
public IloBool	readModelAndSolution(IloModel model, const char * modelFileName, IloSolution solution, const char * solutionFileName) const
public IloBool	readRtti(IloXmlReader reader, IloXmlElement * element) const
public IloBool	readSolution(IloSolution solution, istream & file) const
public IloBool	readSolution(IloSolution solution, const char * fileName) const
public IloBool	readSolutionValue(IloSolution solution, IloXmlElement * root, IloXmlReader reader) const
public void	registerXML(IloTypeIndex index, IloXmlInfo * xmlinfo) const
public void	registerXMLArray(IloXmlInfo * xmlinfo) const
public IloBool	setWriteMode(IloInt mode) const
public void	setWritePrecision(int writePrecision) const
public IloBool	writeModel(const IloModel model, const char * fileName) const
public IloBool	writeModelAndSolution(const IloModel model, const char * modelFileName, const IloSolution solution, const char * solutionFileName) const
public IloBool	writeRtti(const IloRtti * it, IloXmlWriter writer, IloXmlElement * masterElement) const
public IloBool	writeSolution(const IloSolution solution, const char * fileName) const
public void	writeSolutionValue(const IloExtractable it, const IloSolution solution, IloXmlWriter writer) const

Constructors

public **IloXmlContext**(IloEnv env, const char * name=0)

This constructor creates an XML context and makes it part of the environment `env`.

public **IloXmlContext**(IloXmlContextI * impl=0)

This constructor creates a XML context from its implementation object.

Methods

public void **end**()

This member function deletes the invoking XML context.

public IloInt **getChildIdReadError**() const

This member function returns the XML ID of the child unparsed XML element in cases where a problem occurs when reading an `IloModel`.

```
public const char * getChildTagReadError() const
```

This member function returns the XML tag of the child unparsed XML element in cases where a problem occurs when reading an `IloModel`

```
public IloIntArray getIdListReadError() const
```

This member function returns the XML ID list of the unparsed XML elements in cases where a problem occurs when reading an `IloModel`. The list is composed of the tags from the parent to the child elements.

```
public IloXmlContextI * getImpl() const
```

This member function returns the `IloXmlContextI` implementation.

```
public IloInt getParentIdReadError() const
```

This member function returns the XML ID of the parent unparsed XML element in cases where a problem occurs when reading an `IloModel`.

```
public const char * getParentTagReadError() const
```

This member function returns the XML tag of the parent unparsed XML element in cases where a problem occurs when reading an `IloModel`.

```
public IloAnyArray getTagListReadError() const
```

This member function returns the XML tag list of the unparsed XML elements in cases where a problem occurs when reading an `IloModel`. The list is composed of the tags from the parent to the child elements.

```
public const char * getWriteError() const
```

This member function returns the name of the extractable called in cases where a problem occurs when reading an `IloModel`.

```
public int getWritePrecision() const
```

This member function returns the write precision for floats

```
public IloBool readModel(IloModel model, istream & file) const
```

This member function reads `model` from an XML stream.

```
public IloBool readModel(IloModel model, const char * fileName) const
```

This member function reads `model` from the XML file `fileName`.

```
public IloBool readModelAndSolution(IloModel model, const char * modelFileName,
IloSolution solution, const char * solutionFileName) const
```

This member function reads `model` and `solution` from their respective XML files, `modelFileName` and `solutionFileName`.

```
public IloBool readRtti(IloXmlReader reader, IloXmlElement * element) const
```

This member function tries to read all extractables from the XML element.

```
public IloBool readSolution(IloSolution solution, istream & file) const
```

This member function reads `solution` from an XML stream.

Note

This member function only works if a model has already been serialized.

```
public IloBool readSolution(IloSolution solution, const char * fileName) const
```

This member function reads `solution` from the XML file `fileName`.

Note

This member function only works if a model has already been serialized.

```
public IloBool readSolutionValue(IloSolution solution, IloXmlElement * root,
IloXmlReader reader) const
```

This member function reads an `IloSolution` object from an XML element.

```
public void registerXML(IloTypeIndex index, IloXmlInfo * xmlinfo) const
```

This member function registers the serialization class of an extractable with a linked ID, usually its RTTI index. In write mode, the RTTI index is used to catch the correct serialization class.

In read mode, `IloXmlInfo::getTagName` is used to link the correct serialization class to the correct tag.

```
IlpXmlContext context(env);
       context.registerXML(IloAllDiffI::GetTypeIndex(), new (env) IloXmlInfo_AllDiff(context));
```

```
public void registerXMLArray(IloXmlInfo * xmlinfo) const
```

This member function registers the serialization class of an array of extractables with a linked ID.

```
context.registerXMLArray(new (env) IloXmlInfo_SOS2Array(context));
```

```
public IloBool setWriteMode(IloInt mode) const
```

This member function sets the write mode. The write mode can be set to `NoUnknown` or `EvenUnknown`. `NoUnknown` throws an exception if an attempt is made to serialize an unknown extractable. `EvenUnknown` writes a `Unknown` tag with the name of the extractable in a type attribute.

```
public void setWritePrecision(int writePrecision) const
```

This member function sets the write precision for floats. By default, there is no rounding mode on an `IloNum` or an `IloNumArray`. You can also choose the no rounding mode with the `IloNoRoundingMode` constant.

```
public IloBool writeModel(const IloModel model, const char * fileName) const
```

This member function writes `model` to the file `fileName`in XML format.

```
public IloBool writeModelAndSolution(const IloModel model, const char *
modelFileName, const IloSolution solution, const char * solutionFileName) const
```

This member function writes `model` to the file `modelFileName` and `solution` to the file `solutionFileName` in XML format.

```
public IloBool writeRtti(const IloRtti * it, IloXmlWriter writer, IloXmlElement *
masterElement) const
```

This member function writes a specified extractable. It is used from the serialization class of an extractable to write a embedded extractable.

The `IloOr` object calls this method on its constrained `vars`.

See Also: IloXmlInfo::writeRtti

```
public IloBool writeSolution(const IloSolution solution, const char * fileName)
const
```

This member function writes `solution` to the file `fileName` in XML format.

```
public void writeSolutionValue(const IloExtractable it, const IloSolution solution,
IloXmlWriter writer) const
```

This member function writes a specified extractable of a solution in XML. It is used from the serialization class of an extractable to write an embedded extractable.

See Also: IloXmlInfo::writeSolutionValue

Class IloXmlInfo

Definition file: ilconcert/iloxmlabstract.h

The class `IloXmlInfo` allows you to serialize an `IloModel` or an `IloSolution` in XML.

Constructor and Destructor Summary	
public	`IloXmlInfo(IloXmlContextI * context, const char * version=0)`
public	`IloXmlInfo()`

Method Summary	
public IloBool	`checkAttExistence(IloXmlReader reader, IloXmlElement * element, const char * attribute)`
public IloBool	`checkExprExistence(IloXmlReader reader, IloXmlElement * element, const char * attribute, IloInt & id)`
public IloXmlContextI *	`getContext()`
protected IloBool	`getIntValArray(IloXmlReader reader, IloXmlElement * element, IloIntArray & intArray)`
protected IloBool	`getNumValArray(IloXmlReader reader, IloXmlElement * element, IloNumArray & numArray)`
public IloBool	`getRefInChild(IloXmlReader reader, IloXmlElement * element, IloInt & id)`
public virtual const char *	`getTag()`
public virtual IloXmlElement *	`getTagElement(IloXmlWriter writer, const IloRtti * exprI)`
public static const char *	`getTagName()`
protected IloNumVar::Type	`getVarType(IloXmlReader reader, IloXmlElement * element)`
protected const char *	`getVersion()`
protected virtual IloRtti *	`read(IloXmlReader reader, IloXmlElement * element)`
public virtual IloExtractableArray *	`readArrayFromXml(IloXmlReader reader, IloXmlElement * element)`
public virtual IloRtti *	`readFrom(IloXmlReader reader, IloXmlElement * element)`
public virtual IloExtractableI *	`readFromXml(IloXmlReader reader, IloXmlElement * element)`
public IloBool	`readRtti(IloXmlReader reader, IloXmlElement * element)`
public virtual IloBool	`readSolution(IloXmlReader reader, IloSolution solution, IloXmlElement * element)`

protected virtual IloExtractableI *	readXml(IloXmlReader reader, IloXmlElement * element)
protected virtual IloExtractableArray *	readXmlArray(IloXmlReader reader, IloXmlElement * element)
protected IloXmlElement *	setBoolArray(IloXmlWriter writer, const IloBoolArray Array)
public IloXmlElement *	setCommonArrayXml(IloXmlWriter writer, const IloExtractableArray * extractable)
public IloXmlElement *	setCommonValueXml(IloXmlWriter writer, const IloRtti * exprI)
public IloXmlElement *	setCommonXml(IloXmlWriter writer, const IloRtti * exprI)
protected IloXmlElement *	setIntArray(IloXmlWriter writer, const IloIntArray Array)
protected IloXmlElement *	setIntSet(IloXmlWriter writer, const IloIntSet Array)
protected IloXmlElement *	setNumArray(IloXmlWriter writer, const IloNumArray Array)
protected IloXmlElement *	setNumSet(IloXmlWriter writer, const IloNumSet Array)
protected void	setVersion(const char * version)
public void	setXml(IloXmlWriter writer, IloXmlElement * element, const IloRtti * exprI)
public virtual int	write(IloXmlWriter writer, const IloExtractableArray * extractable, IloXmlElement * masterElement)
public virtual IloBool	write(IloXmlWriter writer, const IloRtti * exprI, IloXmlElement * masterElement)
public IloBool	writeExtractable(IloXmlWriter writer, IloXmlElement * element, const IloExtractable extractable, const char * attribute=0)
public virtual IloBool	writeRef(IloXmlWriter writer, const IloRtti * exprI, IloXmlElement * masterElement)
public IloBool	writeRtti(IloXmlWriter writer, IloXmlElement * element, const IloRtti * rtti, const char * attribute=0)
public virtual void	writeSolution(IloXmlWriter writer, const IloSolution solution, const IloExtractable extractable)
public void	writeSolutionValue(IloXmlWriter writer, const IloSolution solution, IloXmlElement * element, const IloRtti * rtti, const char * attribute)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloSOS2Array array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloSOS1Array array)

protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloSemiContVarArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloConstraintArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloRangeArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloNumVarArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloIntSetVarArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloNumExprArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloIntExprArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloBoolVarArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloIntVarArray array)
public virtual IloBool	writeXml(IloXmlWriter writer, const IloExtractableI * exprI, IloXmlElement * masterElement)
public virtual IloBool	writeXmlRef(IloXmlWriter writer, const IloExtractableI * exprI, IloXmlElement * masterElement)

Constructors and Destructors

public **IloXmlInfo**(IloXmlContextI * context, const char * version=0)

This constructor creates an instance of the handle class IloXmlInfo from a pointer to an instance of the undocumented implementation class IloXmlContextI.

public **IloXmlInfo**()

This constructor creates an empty instance of the handle class IloXmlInfo.

Methods

public IloBool **checkAttExistence**(IloXmlReader reader, IloXmlElement * element, const char * attribute)

Given a specified attribute, this member function checks `element` to establish whether the attribute exists. If the attribute does not exist, this member function throws an exception.

You can use this member function to dynamically validate an XML element.

```
public IloBool checkExprExistence(IloXmlReader reader, IloXmlElement * element,
const char * attribute, IloInt & id)
```

Given a specified attribute, this member function checks `element` to establish whether the attribute exists, fills the `id`, and checks in the XML context memory whether an object with this `id` exists.

You can use this member function to dynamically validate an XML element.

Example: in the `read` method of the `IloDiff`, check that the `IdRef` object is already serialized

```
public IloXmlContextI * getContext()
```

This member function returns the related `IloXmlContextI` of the constructor.

```
protected IloBool getIntValArray(IloXmlReader reader, IloXmlElement * element,
IloIntArray & intArray)
```

This member function returns the contained `IloIntArray` in the XML element `element`.

See Also: IloXmlReader::string2IntArray

```
protected IloBool getNumValArray(IloXmlReader reader, IloXmlElement * element,
IloNumArray & numArray)
```

This member function returns the `IloNumArray` in the XML element `element`.

See Also: IloXmlReader::string2NumArray

```
public IloBool getRefInChild(IloXmlReader reader, IloXmlElement * element, IloInt &
id)
```

Given an XML element, this member function checks for the first value `id` or `RefId` in the element and its children.

```
public virtual const char * getTag()
```

This member function returns the related XML tag.

```
public virtual IloXmlElement * getTagElement(IloXmlWriter writer, const IloRtti *
exprI)
```

For backward compatibility with 2.0 and the XML for IloExtractable objects, if this method is not specialized, by default the getTagElement method with IloExtractableI will be called

```
public static const char * getTagName()
```

This static member function returns the linked XML tag of this serialization class.

```
protected IloNumVar::Type getVarType(IloXmlReader reader, IloXmlElement * element)
```

This member function returns the type of an `IloNumVar` - `IloFloat`, `IloInt`, or `IloBool` - in the XML element `element`.

```
protected const char * getVersion()
```

This member function returns the version of the object.

```
protected virtual IloRtti * read(IloXmlReader reader, IloXmlElement * element)
```

This member function reads an `IloRtti` from the given XML element.

This is the method to specialize for each serialization class

For backward compatibility with Concert 2.0 and the XML for IloExtractable objects, by default the method readXml with IloExtractableI will be called

```
public virtual IloExtractableArray * readArrayFromXml(IloXmlReader reader,
IloXmlElement * element)
```

This member function reads an array of `IloRtti*` from the given XML element.

This is the method to specialize when writing a serialization class for an array of extractables.

```
public virtual IloRtti * readFrom(IloXmlReader reader, IloXmlElement * element)
```

This member function reads an `IloRtti` from the given XML element. It asks the XML context to read the extractable in the XML child element using a call to `IloXmlContext::readRtti`; it then calls `IloXmlInfo::readXml`.

For backward compatibility with Concert 2.0 and the XML for IloExtractable objects, by default the method readFromXml with IloExtractableI will be called

```
public virtual IloExtractableI * readFromXml(IloXmlReader reader, IloXmlElement *
element)
```

This member function reads an `IloRtti` from the given XML element. It asks the XML context to read the extractable in the XML child element using a call to `IloXmlContext::readRtti`; it then calls `IloXmlInfo::readXml`.

```
public IloBool readRtti(IloXmlReader reader, IloXmlElement * element)
```

This member function asks the XML context to read the `IloRtti` in the child element and then calls IloXmlInfo::readFromXml to read the parent extractable.

```
public virtual IloBool readSolution(IloXmlReader reader, IloSolution solution,
IloXmlElement * element)
```

This member function reads a variable for `IloSolution` from the XML element `element`.

```
protected virtual IloExtractableI * readXml(IloXmlReader reader, IloXmlElement *
element)
```

This member function reads an `IloRtti` from the given XML element.

This is the method to specialize for each serialization class

```
protected virtual IloExtractableArray * readXmlArray(IloXmlReader reader,
IloXmlElement * element)
```

This member function reads an array of `IloRtti*` from the given XML element.

It is called by the XML context. It first asks the XML context to read from XML child elements using a call to `IloXmlContext::readRtti` and then calls `IloXmlInfo::readArrayFromXml`.

```
protected IloXmlElement * setBoolArray(IloXmlWriter writer, const IloBoolArray
Array)
```

This member function creates an XML element containing the `IloBoolArray`.

See Also: IloXmlWriter::IntArray2String

```
public IloXmlElement * setCommonArrayXml(IloXmlWriter writer, const
IloExtractableArray * extractable)
```

This member function creates a XML element with the common header for `IloExtractable` arrays.

```
public IloXmlElement * setCommonValueXml(IloXmlWriter writer, const IloRtti *
exprI)
```

This member function creates an XML element with the given header for `IloRtti` from `IloSolution`.

```
public IloXmlElement * setCommonXml(IloXmlWriter writer, const IloRtti * exprI)
```

This member function creates an XML element with the common header for `IloRtti`.

```
protected IloXmlElement * setIntArray(IloXmlWriter writer, const IloIntArray Array)
```

This member function creates an XML element containing the `IloIntArray`.

See Also: IloXmlWriter::IntArray2String

```
protected IloXmlElement * setIntSet(IloXmlWriter writer, const IloIntSet Array)
```

This member function creates an XML element containing the `IloIntSet`.

See Also: IloXmlWriter::IntSet2String

```
protected IloXmlElement * setNumArray(IloXmlWriter writer, const IloNumArray Array)
```

This member function creates an XML element containing the `IloNumArray`.

See Also: IloXmlWriter::NumArray2String

```
protected IloXmlElement * setNumSet(IloXmlWriter writer, const IloNumSet Array)
```

This member function creates an XML element containing the `IloNumSet`.

See Also: IloXmlWriter::NumSet2String

```
protected void setVersion(const char * version)
```

This member function sets the version of the object.

```
public void setXml(IloXmlWriter writer, IloXmlElement * element, const IloRtti *
exprI)
```

This member function adds a name attribute and a ID attribute to the XML element.

```
public virtual int write(IloXmlWriter writer, const IloExtractableArray *
extractable, IloXmlElement * masterElement)
```

This member function writes the given `IloExtractableArray` in XML and adds it to the XML document of `writer`. This is the method to specialize when writing a serialization class

```
public virtual IloBool write(IloXmlWriter writer, const IloRtti * exprI,
IloXmlElement * masterElement)
```

This member function writes the `IloRtti` object `exprI` in XML and adds it to the XML document of the `IloXmlWriter` object `writer`.

For backward compatibility with Concert 2.0 and the XML for IloExtractable objects, by default the method writeXml with IloExtractableI will be called

```
public IloBool writeExtractable(IloXmlWriter writer, IloXmlElement * element, const
IloExtractable extractable, const char * attribute=0)
```

See `IloXmlContext::writeRtti(IloXmlWriter,IloXmlElement*,const IloRtti*,const char*)` instead. There is no longer need for the extractable argument.

```
public virtual IloBool writeRef(IloXmlWriter writer, const IloRtti * exprI,
IloXmlElement * masterElement)
```

This member function writes the `IloRtti` object `exprI` in XML as a reference.

For backward compatibility with Concert 2.0 and the XML for IloExtractable objects, by default the method writeXmlRef with IloExtractableI will be called

```
public IloBool writeRtti(IloXmlWriter writer, IloXmlElement * element, const
IloRtti * rtti, const char * attribute=0)
```

This member function writes an embedded extractable. Using the `getId()` method of the extractable, it adds an attribute with the ID in the XML element.

For example, used with `IloDiff`, this member function writes the expression and links it to the XML element via an `IdRef` attribute.

```
// using an IloDiffI* exprI:
    writeRtti(writer, element,
              (IloRtti*)exprI->getExpr1(),
              IloXmlAttributeDef::Expr1Id);
    writeRtti(writer, element,
              (IloRtti*)exprI->getExpr2(),
              IloXmlAttributeDef::Expr2Id);
```

See Also: IloXmlContext::writeRtti

```
public virtual void writeSolution(IloXmlWriter writer, const IloSolution solution,
const IloExtractable extractable)
```

This member function writes the specified extractable `extractable` from the `IloSolutionsolution` in XML format.

```
public void writeSolutionValue(IloXmlWriter writer, const IloSolution solution,
IloXmlElement * element, const IloRtti * rtti, const char * attribute)
```

This member function writes an embedded extractable of a solution in XML. Using the `getId()` method of the extractable, it adds an attribute with the ID in the XML element.

For example, used with `IloDiff`, this member function writes the expression and links it to the XML element via an `IdRef` attribute.

See Also: IloXmlContext::writeSolutionValue

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloSOS2Array array)
```

This member function writes an `IloSOS2Array`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloSOS2s` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloSOS1Array array)
```

This member function writes an `IloSOS1Array`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloSOS1s` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloSemiContVarArray array)
```

This member function writes an `IloSemiContVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloSemiContVars` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloConstraintArray array)
```

This member function writes an `IloConstraintArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloConstraints` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloRangeArray array)
```

This member function writes an `IloRangeArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloRanges` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloNumVarArray array)
```

This member function writes an `IloNumVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloNumVars` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloIntSetVarArray array)
```

This member function writes an `IloIntSetVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloIntSetVars` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloNumExprArray array)
```

This member function writes an `IloNumExprArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloNumExprs` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloIntExprArray array)
```

This member function writes an `IloIntExprArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloIntExprs` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloBoolVarArray array)
```

This member function writes an `IloBoolVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloBoolVars` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloIntVarArray array)
```

This member function writes an `IloIntVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloIntVars` of `array`.

Example using `IloSos` containing an `IloIntVarArray`:

```
// Using an IloSOS1I* exprI;

          this.writeVarArray(writer,
                     element,
                     exprI->getVarArray(),
                     IloXmlAttributeDef::IdRef);
```

This sample adds an `IdRef` attribute on the SOS XML element, creates an XML element containing the `IloIntVarArray` with the list of `IloIntVar` IDs, and creates a list of XML elements for the `IloIntVars`.

```
public virtual IloBool writeXml(IloXmlWriter writer, const IloExtractableI * exprI,
IloXmlElement * masterElement)
```

This member function writes the `IloRtti` object `exprI` in XML and adds it to the XML document of the `IloXmlWriter` object `writer`.

```
public virtual IloBool writeXmlRef(IloXmlWriter writer, const IloExtractableI *
exprI, IloXmlElement * masterElement)
```

This member function writes the `IloRtti` object `exprI` in XML as a reference.

Class IloXmlReader

Definition file: ilconcert/iloreader.h

You can use an instance of `IloXmlReader` to read an `IloModel` or a `IloSolution` in XML format.

Constructor Summary	
public	`IloXmlReader(IloEnv env, const char * fileName=0)`
public	`IloXmlReader(IloXmlReaderI * impl)`

Method Summary	
public IloBool	`checkRttiOfObjectById(IloTypeIndex RTTI, IloRtti * exprI)`
public IloBool	`checkRttiOfObjectById(IloTypeIndex RTTI, IloInt Xml_Id)`
public IloBool	`checkTypeOfObjectById(IloTypeInfo type, IloInt Xml_Id)`
public IloBool	`checkTypeOfObjectById(IloTypeInfo type, IloRtti * exprI)`
public void	`deleteAllocatedMemory(const char * pointer)`
public void	`deleteAllocatedMemory(char * pointer)`
public IloXmlElement *	`findElement(IloXmlElement * root, const char * tag, const char * attribute, const char * value)`
public IloXmlElement *	`findElementByTag(IloXmlElement * element, const char * tag)`
public IloInt	`getChildrenCardinal(IloXmlElement * element)`
public IloEnv	`getEnv()`
public IloEnvI *	`getEnvImpl()`
public IloXmlElement *	`getFirstSubElement(IloXmlElement * element)`
public IloBool	`getIntAttribute(IloXmlElement * element, const char * attribute, IloInt & value)`
public IloBool	`getNumAttribute(IloXmlElement * element, const char * attribute, IloNum & value)`
public IloAny	`getObjectById(IloInt id)`
public IloXmlElement *	`getRoot()`
public IloIntArray *	`getSerialized()`
public IloIntArray *	`getSolutionSerialized()`
public IloBool	`isSerialized(IloInt id)`
public IloBool	`openDocument()`
public const char *	`readAttribute(IloXmlElement * element, const char * attribute)`
public const char *	`readCData(IloXmlElement * element)`
public const char *	`readComment(IloXmlElement * element)`
public const char *	`readData(IloXmlElement * element)`
public const char *	`readText(IloXmlElement * element)`
public void	`setfileName(const char * fileName)`

public IloInt	string2Int(const char * str)
public IloIntArray	string2IntArray(const char * str)
public IloIntRange	string2IntRange(IloXmlElement * element)
public IloIntSet	string2IntSet(const char * str)
public IloNum	string2Num(const char * str)
public IloNumArray	string2NumArray(const char * str)

Constructors

public **IloXmlReader**(IloEnv env, const char * fileName=0)

This constructor creates an `IloXmlReader` object and makes it part of the environment `env`.

The `fileName` is set to `0` by default.

public **IloXmlReader**(IloXmlReaderI * impl)

This constructor creates an XML reader from its implementation object.

Methods

public IloBool **checkRttiOfObjectById**(IloTypeIndex RTTI, IloRtti * exprI)

This method checks the RTTI of the given object.

public IloBool **checkRttiOfObjectById**(IloTypeIndex RTTI, IloInt Xml_Id)

This method checks the RTTI of the object referenced by the identifier `Xml_Id` in the XML. This object must already be serialized.

public IloBool **checkTypeOfObjectById**(IloTypeInfo type, IloInt Xml_Id)

This method checks the TypeInfo of the object referenced by the id in the XML. This object must have been already serialized.

public IloBool **checkTypeOfObjectById**(IloTypeInfo type, IloRtti * exprI)

This method checks the TypeInfo of the given object

public void **deleteAllocatedMemory**(const char * pointer)

This member function frees the memory that has been allocated by the XML reader using, for example, the IloXmlWriter::Int2String member function.

```
public void deleteAllocatedMemory(char * pointer)
```

This member function frees the memory that has been allocated by the XML reader using, for example, the IloXmlWriter::Int2String member function.

```
public IloXmlElement * findElement(IloXmlElement * root, const char * tag, const
char * attribute, const char * value)
```

This member function examines the XML element `root` to identify the XML child element denoted by `tag`, `attribute`, and `value`.

```
public IloXmlElement * findElementByTag(IloXmlElement * element, const char * tag)
```

This member function examines the XML element `element` to identify the XML child element denoted by `tag`.

```
public IloInt getChildrenCardinal(IloXmlElement * element)
```

This member function counts the number of child elements of the XML element `element`.

```
public IloEnv getEnv()
```

This member function gets the `IloEnv` of the object.

```
public IloEnvI * getEnvImpl()
```

This member function gets the implementation of the `IloEnv` of the object.

```
public IloXmlElement * getFirstSubElement(IloXmlElement * element)
```

This member function gets the first child in the XML element `element`.

```
public IloBool getIntAttribute(IloXmlElement * element, const char * attribute,
IloInt & value)
```

This member function checks the existence of `attribute` in the XML element `element` and converts it to an `IloInt`.

```
public IloBool getNumAttribute(IloXmlElement * element, const char * attribute,
IloNum & value)
```

This member function checks the existence of `attribute` in the XML element `element` and converts it to an `IloNum`.

```
public IloAny getObjectById(IloInt id)
```

This member function gets the already serialized object of the given identifier `id`.

```
IloDiff Diff(reader.getEnv(),
                IloExpr((IloNumExprI*)reader.getObjectById(IdExpr1)),
                IloExpr((IloNumExprI*)reader.getObjectById(IdExpr2)),
                reader.readAttribute(element, IloXmlAttributeDef::Name));
```

The sample code creates a `IloDiff` from a XML element referencing its two expressions with the attributes `IdRef1` and `IdRef2`.

```
public IloXmlElement * getRoot()
```

This member function gets the XML root, that is, the XML document without the header.

```
public IloIntArray * getSerialized()
```

This member function gets the IDs of the serialized extractables and the unique IDs of the array of extractables that were serialized from the model.

```
public IloIntArray * getSolutionSerialized()
```

This member function gets the IDs of the serialized extractables and the unique IDs of the array of extractables that were serialized from the solution.

```
public IloBool isSerialized(IloInt id)
```

This member function checks whether the extractable with the ID `id` in the model has already been serialized.

```
public IloBool openDocument()
```

This member function opens the XML document specified in the constructor or with the `setFileName` method.

```
public const char * readAttribute(IloXmlElement * element, const char * attribute)
```

This member function returns the value of the `attribute` in the XML element `element`.

```
public const char * readCData(IloXmlElement * element)
```

This member function reads the CDATA of the XML element `element`.

```
public const char * readComment(IloXmlElement * element)
```

This member function returns the value of the comment in the XML element `element`.

```
public const char * readData(IloXmlElement * element)
```

This member function reads the data of the XML element `element`.

```
public const char * readText(IloXmlElement * element)
```

This member function returns the value of the text contained in the XML element `element`, independently of its origin (data or CDATA).

```
public void setfileName(const char * fileName)
```

This member function sets `fileName` as the file from which to read the XML.

```
public IloInt string2Int(const char * str)
```

This member function converts `str` into an `IloInt`.

```
public IloIntArray string2IntArray(const char * str)
```

This member function converts `str` into an `IloIntArray`.

```
public IloIntRange string2IntRange(IloXmlElement * element)
```

This member function converts `str` into an `IloIntRange`.

```
public IloIntSet string2IntSet(const char * str)
```

This member function converts `str` into an `IloIntSet`.

```
public IloNum string2Num(const char * str)
```

This member function converts `str` into an `IloNum`.

```
public IloNumArray string2NumArray(const char * str)
```

This member function converts `str` into an `IloNumArray`.

Class IloXmlWriter

Definition file: ilconcert/ilowriter.h

You can use an instance of `IloXmlWriter` to serialize an `IloModel` or an `IloSolution` in XML.

Constructor Summary	
public	`IloXmlWriter(IloEnv env, const char * rootTag, const char * fileName=0)`
public	`IloXmlWriter(IloXmlWriterI * impl)`

Method Summary	
public void	`addAttribute(IloXmlElement * element, const char * attribute, const char * value)`
public void	`addCData(IloXmlElement * element, const char * CData)`
public void	`addComment(IloXmlElement * element, const char * comment)`
public void	`addElement(IloXmlElement * element)`
public void	`addSubElement(IloXmlElement * element, IloXmlElement * subElement)`
public void	`addText(IloXmlElement * element, const char * text)`
public IloXmlElement *	`createElement(const char * element)`
public void	`deleteAllocatedMemory(const char * pointer)`
public void	`deleteAllocatedMemory(char * pointer)`
public IloEnv	`getEnv()`
public IloEnvI *	`getEnvImpl()`
public const char *	`getfileName()`
public IloXmlElement *	`getRoot()`
public IloIntArray *	`getSerialized()`
public IloIntArray *	`getSolutionSerialized()`
public const char *	`Int2String(const IloInt number)`
public const char *	`IntArray2String(const IloIntArray intArray)`
public const char *	`IntSet2String(const IloIntSet intSet)`
public IloBool	`isSerialized(IloInt id)`
public IloBool	`isSolutionSerialized(IloInt id)`
public const char *	`Num2String(const IloNum number)`
public const char *	`NumArray2String(const IloNumArray numArray)`
public const char *	`NumSet2String(const IloNumSet numSet)`
public void	`setfileName(const char * fileName)`
public IloInt	`string2Int(const char * str)`
public IloBool	`writeDocument()`

Constructors

`public` **`IloXmlWriter`**`(IloEnv env, const char * rootTag, const char * fileName=0)`

This constructor creates an `IloXmlWriter` object and makes it part of the environment `env`.

The `fileName` is set to `0` by default.

`public` **`IloXmlWriter`**`(IloXmlWriterI * impl)`

This constructor creates a XML writer object from its implementation object.

Methods

`public void` **`addAttribute`**`(IloXmlElement * element, const char * attribute, const char * value)`

This member function adds an attribute of the specified value to the XML element.

`public void` **`addCData`**`(IloXmlElement * element, const char * CData)`

This member function adds a CDATA section to the XML element `element`.

`public void` **`addComment`**`(IloXmlElement * element, const char * comment)`

This member function adds `comment` to the XML element `element`.

`public void` **`addElement`**`(IloXmlElement * element)`

This member function adds the XML element `element` to the end of the XML.

`public void` **`addSubElement`**`(IloXmlElement * element, IloXmlElement * subElement)`

This member function adds a child element, `subElement`, to the XML element `element`.

`public void` **`addText`**`(IloXmlElement * element, const char * text)`

This member function adds `text` to the specified element.

`public IloXmlElement *` **`createElement`**`(const char * element)`

This member function creates an empty element with the given tag, `element`.

```
public void deleteAllocatedMemory(const char * pointer)
```

This member function frees the memory that has been allocated by the XML reader using, for example, the IloXmlWriter::Int2String member function.

```
public void deleteAllocatedMemory(char * pointer)
```

This member function frees the memory that has been allocated by the XML reader using, for example, the IloXmlWriter::Int2String member function.

```
public IloEnv getEnv()
```

This member function gets the `IloEnv` of the object.

```
public IloEnvI * getEnvImpl()
```

This member function gets the implementation of the `IloEnv` of the object.

```
public const char * getfileName()
```

This member function returns the name of the XML file

```
public IloXmlElement * getRoot()
```

This member function gets the root XML element of the XML document.

```
public IloIntArray * getSerialized()
```

This member function gets the IDs of the serialized objects of an `IloModel`.

```
public IloIntArray * getSolutionSerialized()
```

This member function gets the IDs of the serialized objects of an `IloSolution`.

```
public const char * Int2String(const IloInt number)
```

This member function converts the `IloInt` object `number` into a string, `const char*`.

```
public const char * IntArray2String(const IloIntArray intArray)
```

This member function converts the `IloIntArray` object `intArray` into a string, `const char*`.

`public const char * IntSet2String(const IloIntSet intSet)`

This member function converts the `IloIntSet` object `intSet` into a string, `const char*`.

`public IloBool isSerialized(IloInt id)`

This member function checks whether an object has been serialized.

`public IloBool isSolutionSerialized(IloInt id)`

This member function checks whether a solution object has already been serialized.

`public const char * Num2String(const IloNum number)`

This member function converts the `IloNum` object `number` into a string, `const char*`.

`public const char * NumArray2String(const IloNumArray numArray)`

This member function converts the `IloNumArray` object `numArray` into a string, `const char*`.

`public const char * NumSet2String(const IloNumSet numSet)`

This member function converts the `IloNumSet` object `numSet` into a string, `const char*`.

`public void setfileName(const char * fileName)`

This member function specifies `fileName` as the name of the XML file.

`public IloInt string2Int(const char * str)`

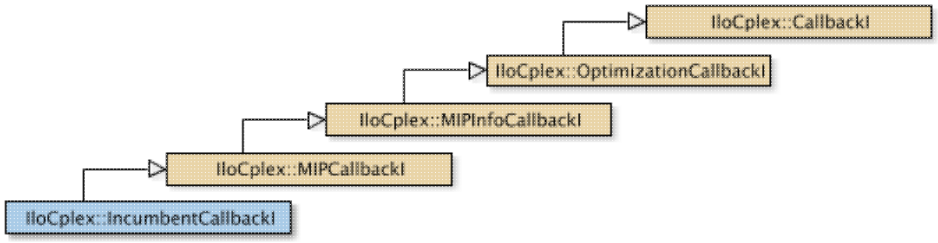
This member function converts `str` into an `IloInt`.

`public IloBool writeDocument()`

This member function outputs the XML to the file specified in the constructor or using the `setFileName` method. If null, this member function outputs on the `cout io`.

Class IloCplex::IncumbentCallbackI

Definition file: ilcplex/ilocplexi.h



Note

This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

This callback is called whenever a new potential incumbent is found during branch-and-cut searches. It allows you to analyze the proposed incumbent and optionally reject it. In this case, CPLEX will continue the branch-and-cut search. This callback is thus typically combined with a branch callback that instructs CPLEX how to branch on a node after it has found a potential incumbent and thus considered the node solution to be integer feasible.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI, IloCplex::OptimizationCallbackI, ILOINCUMBENTCALLBACK0

Method Summary	
public NodeData *	getNodeData() const
public NodeId	getNodeId() const
public IloNum	getObjValue() const
public IloNum	getSlack(const IloRange rng) const
public void	getSlacks(IloNumArray val, const IloRangeArray con) const
public IloNum	getValue(const IloIntVar var) const
public IloNum	getValue(const IloNumVar var) const
public IloNum	getValue(const IloExprArg expr) const
public void	getValues(IloNumArray val, const IloIntVarArray vars) const
public void	getValues(IloNumArray val, const IloNumVarArray vars) const
public void	reject()

Inherited Methods from `MIPCallbackI`
getNcliques, getNcovers, getNcuts, getNdisjunctiveCuts, getNflowCovers, getNflowPaths, getNfractionalCuts, getNGUBcovers, getNimpliedBounds, getNMIRs, getNzeroHalfCuts, getObjCoef, getObjCoef, getObjCoefs, getObjCoefs, getUserThreads

Inherited Methods from `MIPInfoCallbackI`
getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum,

getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent

Inherited Methods from `OptimizationCallbackI`
getModel, getNcols, getNQCs, getNrows

Inherited Methods from `CallbackI`
abort, duplicateCallback, getEndTime, getEnv, main

Methods

```
public NodeData * getNodeData() const
```

This method retrieves the `NodeData` object that may have previously been assigned to the current node by the user with the method `IloCplex::BranchCallbackI::makeBranch`. If no data object has been assigned to the current node, `0` will be returned.

```
public NodeId getNodeId() const
```

This method returns the `NodeId` of the current node.

```
public IloNum getObjValue() const
```

This method returns the query objective value of the potential incumbent.

If you need the object representing the objective itself, consider the method `IloCplex::getObjective` instead.

```
public IloNum getSlack(const IloRange rng) const
```

This method returns the slack value for the range specified by `rng` for the potential incumbent.

```
public void getSlacks(IloNumArray val, const IloRangeArray con) const
```

This method puts the slack value for each range in the array of ranges `con` into the corresponding element of the array `val` for the potential incumbent. For this CPLEX resizes array `val` to match the size of array `con`.

```
public IloNum getValue(const IloIntVar var) const
```

This method returns the query value of the variable `var` in the potential incumbent solution.

```
public IloNum getValue(const IloNumVar var) const
```

This method returns the value of the variable `var` in the potential incumbent solution.

```
public IloNum getValue(const IloExprArg expr) const
```

This method returns the value of the `expr` for the potential incumbent solution.

```
public void getValues(IloNumArray val, const IloIntVarArray vars) const
```

This method returns the query values of the variables in the array `vars` in the potential incumbent solution and copies them to `val`. CPLEX automatically resizes the array `val` to match the size of the array `vars`.

```
public void getValues(IloNumArray val, const IloNumVarArray vars) const
```

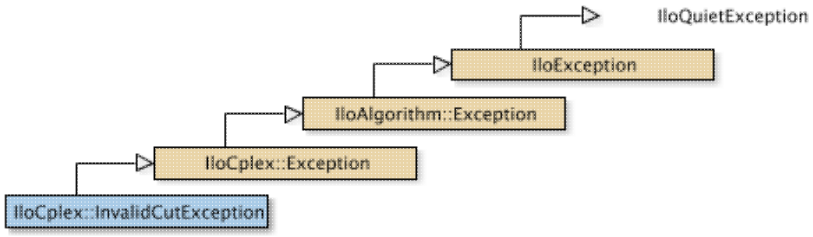
This method returns the query values of the variables in the array `vars` in the potential incumbent solution and copies them to `val`. CPLEX automatically resizes the array `val` to match the length of the array `vars`.

```
public void reject()
```

This method rejects the proposed incumbent.

Class IloCplex::InvalidCutException

Definition file: ilcplex/ilocplexi.h



An instance of this exception is thrown by `IloCplex` when an an attempt is made to add a malformed cut. An example of a malformed cut is one that uses variables that have not been extracted or a cut that is defined with an expression that is not linear.

Method Summary	
`public IloConstraint`	`getCut() const`

Inherited Methods from `Exception`
`getStatus`

Inherited Methods from `IloException`
`end, getMessage`

Methods

`public IloConstraint` **`getCut`**`() const`

Returns the invalid cut that triggered the invoking exception.

Class IloModel::Iterator

Definition file: ilconcert/ilomodel.h

Nested class of iterators to traverse the extractable objects in a model.
An instance of this nested class is an iterator capable of traversing the extractable objects in a model.

An iterator of this class differs from one created by the template `IloIterator`. Instances of `IloIterator` traverse all the extractable objects of a given class (specified by `E` in the template) within a given environment (an instance of `IloEnv`), whether or not those extractable objects have been explicitly added to a model. Instances of `IloModel::Iterator` traverse only those extractable objects that have explicitly been added to a given model (an instance of `IloModel`).

See Also: IloIterator, IloModel

Constructor Summary	
public	Iterator(const IloModel model)

Method Summary	
public IloBool	ok() const
public IloExtractable	operator*()
public void	operator++()

Constructors

`public` **`Iterator`**`(const IloModel model)`

This constructor creates an iterator to traverse the extractable objects in the model specified by `model`.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IloFalse`.

`public IloExtractable` **`operator*`**`()`

This operator returns the current extractable object, the one to which the invoking iterator points.

`public void` **`operator++`**`()`

This operator advances the iterator to point to the next extractable object in the model.

Class IloSolution::Iterator

Definition file: ilconcert/ilosolution.h

It allows you to traverse the variables in a solution.
`Iterator` is a class nested in the class `IloSolution`. It allows you to traverse the variables in a solution. The iterator scans the objects in the same order as they were added to the solution.

This iterator is not robust. If the variable at the current position is deleted from the solution being iterated over, the behavior of this iterator afterward is undefined.

- `iter` can be safely used after the following code has executed:

```
IloExtractable elem = *iter;

++iter;

solution.remove(elem);
```

- `iter` cannot be safely used after the following code has executed:

```
solution.remove(*iter); // bad idea

++iter;
```

See Also: IloIterator, IloSolution

Constructor Summary	
public	`Iterator(IloSolution solution)`

Method Summary	
public IloBool	`ok() const`
public IloExtractable	`operator*() const`
public Iterator &	`operator++()`

Constructors

`public` **`Iterator`**`(IloSolution solution)`

This constructor creates an iterator to traverse the variables of `solution`. The iterator traverses variables in the same order they were added to `solution`.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all variables have been scanned by the iterator.

```
public IloExtractable operator*() const
```

This operator returns the extractable object corresponding to the variable located at the current iterator position. If all variables have been scanned, this operator returns an empty handle.

```
public Iterator & operator++()
```

This operator moves the iterator to the next variable in the solution.

Class ParameterSet::Iterator

Definition file: ilcplex/ilocplexi.h

An instance of this nested class is an iterator that traverses a set of parameters.

The class includes operators to point to the current parameter in the set and to advance to the next parameter in the set, and a method `ok` to verify that the iterator is still within the set.

Constructor Summary	
public	Iterator(IloCplex::ParameterSet)

Method Summary	
public bool	ok() const
public IloCplex::Parameter	operator*() const
public Iterator	operator++(int)
public Iterator &	operator++()

Constructors

`public` **`Iterator`**`(IloCplex::ParameterSet)`

Constructs an iterator capable of traversing the parameters in the designated parameter set.

Methods

`public bool` **`ok`**`() const`

Returns true if the iterator points to a valid element of the the parameter set, and false otherwise.

`public IloCplex::Parameter` **`operator*`**`() const`

Returns the parameter to which the iterator currently points.

`public Iterator` **`operator++`**`(int)`

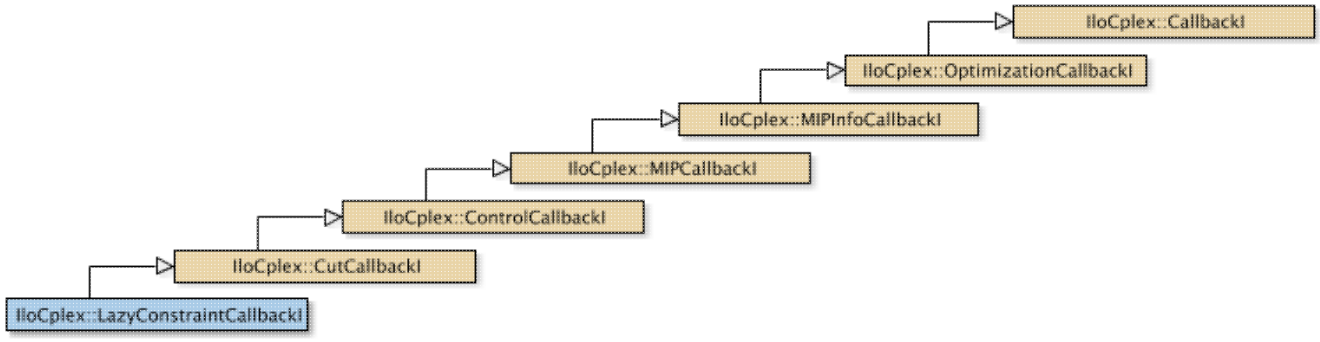
Advances the iterator to the next element of the parameter set.

`public Iterator &` **`operator++`**`()`

Advances the iterator to the next element of the parameter set.

Class IloCplex::LazyConstraintCallbackI

Definition file: ilcplex/ilocplexi.h



Note

This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::LazyConstraintCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a MIP while generating lazy constraints. `IloCplex` calls the user-written callback after solving each node LP exactly like `IloCplex::CutCallbackI`. In fact, this callback is exactly equivalent to `IloCplex::CutCallbackI` but offers a name more consistently pointing out the difference between lazy constraints and user cuts.

Inherited Methods from `CutCallbackI`
`add, addLocal`

Inherited Methods from `ControlCallbackI`
`getDownPseudoCost, getDownPseudoCost, getFeasibilities, getFeasibilities, getFeasibility, getFeasibility, getFeasibility, getFeasibility, getLB, getLB, getLBs, getLBs, getNodeData, getObjValue, getSlack, getSlacks, getUB, getUB, getUBs, getUBs, getUpPseudoCost, getUpPseudoCost, getValue, getValue, getValue, getValues, getValues, isSOSFeasible, isSOSFeasible`

Inherited Methods from `MIPCallbackI`
`getNcliques, getNcovers, getNcuts, getNdisjunctiveCuts, getNflowCovers, getNflowPaths, getNfractionalCuts, getNGUBcovers, getNimpliedBounds, getNMIRs, getNzeroHalfCuts, getObjCoef, getObjCoef, getObjCoefs, getObjCoefs, getUserThreads`

Inherited Methods from `MIPInfoCallbackI`
`getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent`

Inherited Methods from `OptimizationCallbackI`
`getModel, getNcols, getNQCs, getNrows`

Inherited Methods from `CallbackI`
`abort, duplicateCallback, getEndTime, getEnv, main`

Class IloExpr::LinearIterator

Definition file: ilconcert/iloexpression.h

An iterator over the linear part of an expression.
An instance of the nested class IloExpr::LinearIterator is an iterator that traverses the linear part of an expression.

Example

Start with an expression that contains both linear and non linear terms:

```
IloExpr e = 2*x + 3*y + cos(x);
```

Now define a linear iterator for the expression:

```
IloExpr::LinearIterator it = e.getLinearIterator();
```

That constructor creates a linear iterator initialized on the first linear term in `e`, that is, the term `(2*x)`. Consequently, a call to the member function `ok` returns `IloTrue`.

```
it.ok(); // returns IloTrue
```

A call to the member function `getCoef` returns the coefficient of the current linear term.

```
it.getCoef(); // returns 2 from the term (2*x)
```

Likewise, the member function `getVar` returns the handle of the variable of the current linear term.

```
it.getVar(); // returns handle of x from the term (2*x)
```

A call to the `operator++` at this point advances the iterator to the next linear term, `(3*y)`. The iterator ignores nonlinear terms in the expression.

```
++it; // goes to next linear term (3*y)
it.ok(); // returns IloTrue
it.getCoef(); // returns 3 from the term (3*y)
it.getVar(); // returns handle of y from the term (3*y)
++it; // goes to next linear term, if there is one in the expression
it.ok(); // returns IloFalse because there is no linear term
```

Method Summary	
public IloNum	getCoef() const
public IloNumVar	getVar() const
public IloBool	ok() const
public void	operator++()

Methods

`public IloNum` **`getCoef`**`() const`

This member function returns the coefficient of the current term.

```
public IloNumVar getVar() const
```

This member function returns the variable of the current term.

```
public IloBool ok() const
```

This member function returns `IloTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IloFalse`.

```
public void operator++()
```

This operator advances the iterator to point to the next term of the linear part of the expression.

Class IloCsvReader::LineIterator

Definition file: ilconcert/ilocsvreader.h

Line-iterator for csv readers.
`LineIterator` is a nested class of the class `IloCsvReader`. It is to be used only with csv reader objects built to read a unique-table data file.

`IloCsvReader::LineIterator` allows you to step through all the lines of the csv data file (except blank lines and commented lines) on which the csv reader was created.

Constructor and Destructor Summary	
public	`LineIterator()`
public	`LineIterator(IloCsvReader csv)`

Method Summary	
public IloBool	`ok() const`
public IloCsvLine	`operator*() const`
public LineIterator &	`operator++()`

Constructors and Destructors

`public` **`LineIterator`**`()`

This constructor creates an empty `LineIterator` object. This object must be assigned before it can be used.

`public` **`LineIterator`**`(IloCsvReader csv)`

This constructor creates an iterator to traverse all the lines in the csv data file on which the csv reader `csv` was created.

The iterator does not traverse blank lines and commented lines.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one.

It returns `IloFalse` if the iterator reaches the end of the table.

`public IloCsvLine` **`operator*`**`() const`

This operator returns the current instance of `IloCsvLine` (representing the current line in the csv file); the one to which the invoking iterator points.

```
public LineIterator & operator++()
```

This left-increment operator shifts the current position of the iterator to the next instance of `IloCsvLine` representing the next line in the file.

Class IloCsvTableReader::LineIterator

Definition file: ilconcert/ilocsvreader.h

Line-iterator for csv table readers.
`LineIterator` is a nested class of the class `IloCsvTableReader`. It allows you to step through all the lines of a table from a csv data file (except blank lines and commented lines) on which the table csv reader was created.

Constructor and Destructor Summary	
`public`	`LineIterator()`
`public`	`LineIterator(IloCsvTableReader csv)`

Method Summary	
`public IloBool`	`ok() const`
`public IloCsvLine`	`operator*() const`
`public LineIterator &`	`operator++()`

Constructors and Destructors

`public` **`LineIterator`**`()`

This constructor creates an empty `LineIterator` object.

This object must be assigned before it can be used.

`public` **`LineIterator`**`(IloCsvTableReader csv)`

This constructor creates an iterator to traverse all the lines in the table csv data file on which the csv reader `csv` was created.

The iterator does not traverse blank lines and commented lines.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one.

It returns `IloFalse` if the iterator reaches the end of the table.

`public IloCsvLine` **`operator*`**`() const`

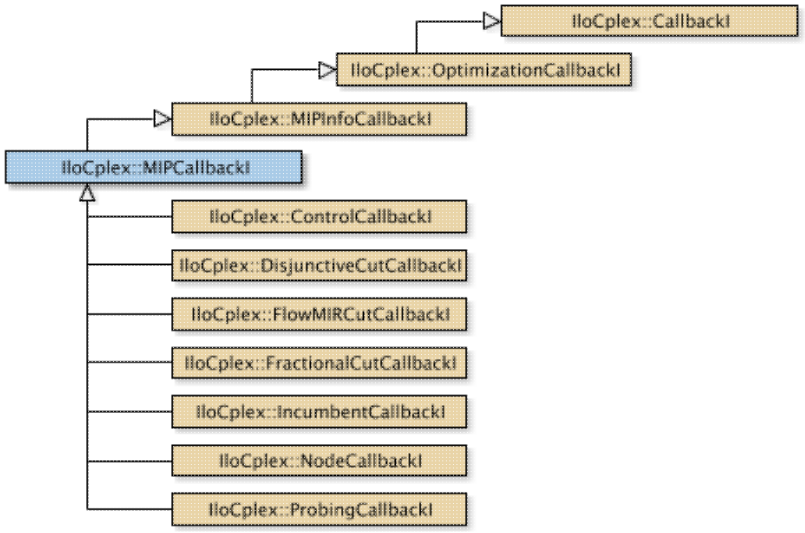
This operator returns the current instance of `IloCsvLine` (representing the current line in the csv file); the one to which the invoking iterator points.

```
public LineIterator & operator++()
```

This left-increment operator shifts the current position of the iterator to the next instance of `IloCsvLine` representing the next line in the file.

Class IloCplex::MIPCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::MIPCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer program (MIP). `IloCplex` calls the user-written callback prior to solving each node in branch-and-cut search.

User-written callbacks of this class or any of its subclasses are **not** compatible with MIP dynamic search. If you are looking for support for callbacks compatible with dynamic search, consider the class `IloCplex::MIPInfoCallbackI` instead.

This class offers member functions for accessing an incumbent solution and its objective value from a user-written callback. It also offers methods for accessing priority orders and statistical information, such as the number of cuts. Methods are also available to query the number of generated cuts for each type of cut CPLEX generates. See the *CPLEX User's Manual* for more information about cuts.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown. For example, if there is no incumbent, the methods that query about incumbents will throw an exception.

This class also provides the common application programming interface (API) for these callback classes:

- `IloCplex::NodeCallbackI`
- `IloCplex::IncumbentCallbackI`
- `IloCplex::DisjunctiveCutCallbackI`
- `IloCplex::FlowMIRCutCallbackI`
- `IloCplex::FractionalCutCallbackI`
- `IloCplex::ProbingCallbackI`
- `IloCplex::CutCallbackI`
- `IloCplex::BranchCallbackI`
- `IloCplex::HeuristicCallbackI`
- `IloCplex::SolveCallbackI`

See Also: IloCplex, IloCplex::BranchCallbackI, IloCplex::Callback, IloCplex::CallbackI, IloCplex::CutCallbackI, IloCplex::DisjunctiveCutCallbackI, IloCplex::FlowMIRCutCallbackI, IloCplex::FractionalCutCallbackI, IloCplex::HeuristicCallbackI, IloCplex::IncumbentCallbackI, IloCplex::NodeCallbackI, IloCplex::OptimizationCallbackI, IloCplex::ProbingCallbackI, IloCplex::SolveCallbackI, ILOMIPCALLBACK0

Constructor Summary	
protected	MIPCallbackI(IloEnv env)

Method Summary	
public IloInt	getNcliques() const
public IloInt	getNcovers() const
public IloInt	getNcuts(IloCplex::CutType which) const
public IloInt	getNdisjunctiveCuts() const
public IloInt	getNflowCovers() const
public IloInt	getNflowPaths() const
public IloInt	getNfractionalCuts() const
public IloInt	getNGUBcovers() const
public IloInt	getNimpliedBounds() const
public IloInt	getNMIRs() const
public IloInt	getNzeroHalfCuts() const
public IloNum	getObjCoef(const IloIntVar var) const
public IloNum	getObjCoef(const IloNumVar var) const
public void	getObjCoefs(IloNumArray val, const IloIntVarArray vars) const
public void	getObjCoefs(IloNumArray val, const IloNumVarArray vars) const
public IloInt	getUserThreads() const

Inherited Methods from **MIPInfoCallbackI**
getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent

Inherited Methods from **OptimizationCallbackI**
getModel, getNcols, getNQCs, getNrows

Inherited Methods from **CallbackI**
abort, duplicateCallback, getEndTime, getEnv, main

Inner Class
MIPCallbackI::NodeData

Constructors

protected **MIPCallbackI**(IloEnv env)

This constructor creates a callback for use in an application that uses an instance of IloCplex to solve a mixed integer program (MIP).

Methods

`public IloInt` **`getNcliques`**`() const`

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of clique cuts that have been added to the model so far during the current optimization.

`public IloInt` **`getNcovers`**`() const`

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of cover cuts that have been added to the model so far during the current optimization.

`public IloInt` **`getNcuts`**`(IloCplex::CutType which) const`

Returns the total number of cuts of the type `which` that CPLEX has added to the model so far during the current optimization.

`public IloInt` **`getNdisjunctiveCuts`**`() const`

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of disjunctive cuts that have been added to the model so far during the current optimization.

`public IloInt` **`getNflowCovers`**`() const`

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of flow cover cuts that have been added to the model so far during the current optimization.

`public IloInt` **`getNflowPaths`**`() const`

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of flow path cuts that have been added to the model so far during the current optimization.

`public IloInt` **`getNfractionalCuts`**`() const`

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of fractional cuts that have been added to the model so far during the current optimization.

```
public IloInt getNGUBcovers() const
```

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of GUB cover cuts that have been added to the model so far during the current optimization.

```
public IloInt getNimpliedBounds() const
```

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of implied bound cuts that have been added to the model so far during the current optimization.

```
public IloInt getNMIRs() const
```

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of MIR cuts that have been added to the model so far during the current optimization.

```
public IloInt getNzeroHalfCuts() const
```

This method has been **deprecated**. Instead, use the method `IloCplex::getNcuts` with an argument specifying the type of cut to count.

Returns the total number of zero-half cuts that have been added to the model so far during the current optimization.

```
public IloNum getObjCoef(const IloIntVar var) const
```

Returns the linear objective coefficient for `var` in the model currently being solved.

```
public IloNum getObjCoef(const IloNumVar var) const
```

Returns the linear objective coefficient for `var` in the model currently being solved.

```
public void getObjCoefs(IloNumArray val, const IloIntVarArray vars) const
```

Puts the linear objective coefficient of each of the variables in the array `vars` into the corresponding element of the array `vals`.

```
public void getObjCoefs(IloNumArray val, const IloNumVarArray vars) const
```

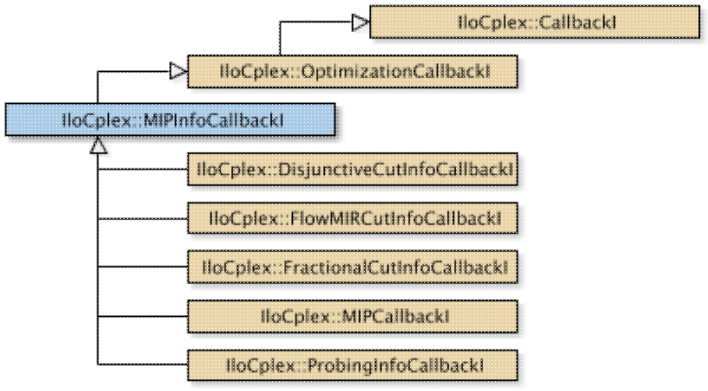
Puts the linear objective coefficient of each of the variables in the array `vars` into the corresponding element of the array `vals`.

```
public IloInt getUserThreads() const
```

Returns the total number of parallel threads currently running.

Class IloCplex::MIPInfoCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::MIPInfoCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer program (MIP). `IloCplex` calls the user-written callback regularly during the branch-and-cut search.

User-written callbacks of this class are compatible with MIP dynamic search.

This class offers methods for accessing an incumbent solution and its objective value from a user-written callback. It also offers methods for accessing priority orders and progress information, such as the number of nodes solved.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made in a user-written callback to access information not available to an instance of this class, an exception is raised. For example, if there is no incumbent, the methods that query about incumbents will throw an exception.

This class also provides the common application programming interface (API) for these callback classes:

- `IloCplex::DisjunctiveCutInfoCallbackI`
- `IloCplex::FlowMIRCutInfoCallbackI`
- `IloCplex::FractionalCutInfoCallbackI`
- `IloCplex::ProbingInfoCallbackI`

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::DisjunctiveCutInfoCallbackI, IloCplex::FlowMIRCutInfoCallbackI, IloCplex::FractionalCutInfoCallbackI, IloCplex::OptimizationCallbackI, IloCplex::ProbingInfoCallbackI, ILOMIPINFOCALLBACK0

Constructor Summary	
`protected`	`MIPInfoCallbackI(IloEnv env)`

Method Summary	
`public IloNum`	`getBestObjValue() const`
`public IloNum`	`getCutoff() const`
`public IloCplex::BranchDirection`	`getDirection(const IloIntVar var) const`
`public IloCplex::BranchDirection`	`getDirection(const IloNumVar var) const`
`public IloNum`	`getIncumbentObjValue() const`

public IloNum	getIncumbentSlack(const IloRange rng) const
public void	getIncumbentSlacks(IloNumArray vals, const IloRangeArray cons) const
public IloNum	getIncumbentValue(const IloIntVar var) const
public IloNum	getIncumbentValue(const IloNumVar var) const
public void	getIncumbentValues(IloNumArray val, const IloIntVarArray vars) const
public void	getIncumbentValues(IloNumArray val, const IloNumVarArray vars) const
public IloNum	getMIPRelativeGap() const
public IloInt	getMyThreadNum() const
public IloInt	getNiterations() const
public IloInt	getNnodes() const
public IloInt	getNremainingNodes() const
public IloNum	getPriority(const IloIntVar sos) const
public IloNum	getPriority(const IloNumVar sos) const
public IloBool	hasIncumbent() const

Inherited Methods from `OptimizationCallbackI`
getModel, getNcols, getNQCs, getNrows

Inherited Methods from `CallbackI`
abort, duplicateCallback, getEndTime, getEnv, main

Constructors

protected **MIPInfoCallbackI**(IloEnv env)

This constructor creates a callback for use in an application that uses an instance of `IloCplex` to solve a mixed integer program (MIP).

Methods

public IloNum **getBestObjValue**() const

This method accesses the currently best known bound of all the remaining open nodes in a branch-and-cut tree.

It is computed for a minimization problem as the minimum objective function value of all remaining unexplored nodes. Similarly, it is computed for a maximization problem as the maximum objective function value of all remaining unexplored nodes.

For a regular MIP optimization, this value is also the best known bound on the optimal solution value of the MIP problem. In fact, when a problem has been solved to optimality, this value matches the optimal solution value.

However, for the method `populate`, the value can also exceed the optimal solution value if CPLEX has already solved the model to optimality but continues to search for additional solutions.

public IloNum **getCutoff**() const

Returns the current cutoff value.

An instance of `IloCplex` uses the cutoff value (the value of the objective function of the subproblem at a node in the search tree) to decide when to prune nodes from the search tree (that is, when to cut off that node and discard the nodes beyond it). The cutoff value is updated whenever a new incumbent is found.

```
public IloCplex::BranchDirection getDirection(const IloIntVar var) const
```

This method returns the branch direction previously assigned to variable `var` with the method `IloCplex::setDirection` or `IloCplex::setDirections`. If no direction has been assigned, `IloCplex::BranchGlobal` will be returned.

```
public IloCplex::BranchDirection getDirection(const IloNumVar var) const
```

This method returns the branch direction previously assigned to variable `var` with the method `IloCplex::setDirection` or `IloCplex::setDirections`. If no direction has been assigned, `IloCplex::BranchGlobal` will be returned.

```
public IloNum getIncumbentObjValue() const
```

Returns the value of the objective function of the incumbent solution (that is, the best integer solution found so far) at the time the invoking callback is called by an instance of `IloCplex` while solving a MIP. If there is no incumbent, this method throws an exception.

```
public IloNum getIncumbentSlack(const IloRange rng) const
```

This method returns the slack value for the range specified by `rng` for the incumbent. If there is no incumbent, this method throws an exception.

```
public void getIncumbentSlacks(IloNumArray vals, const IloRangeArray cons) const
```

This method puts the slack value for each range in the array of ranges `cons` into the corresponding element of the array `vals` for the incumbent. CPLEX resizes array `vals` to match the size of array `cons`. If there is no incumbent, this method throws an exception.

```
public IloNum getIncumbentValue(const IloIntVar var) const
```

Returns the solution value of `var` in the incumbent solution at the time the invoking callback is called by an instance of `IloCplex` while solving a MIP. If there is no incumbent, this method throws an exception.

```
public IloNum getIncumbentValue(const IloNumVar var) const
```

Returns the solution value of `var` in the incumbent solution at the time the invoking callback is called by an instance of `IloCplex` while solving a MIP. If there is no incumbent, this method throws an exception.

```
public void getIncumbentValues(IloNumArray val, const IloIntVarArray vars) const
```

Returns the value of each variable in the array `vars` with respect to the current incumbent solution, and it puts those values into the corresponding array `vals`. If there is no incumbent, this method throws an exception.

```
public void getIncumbentValues(IloNumArray val, const IloNumVarArray vars) const
```

Returns the value of each variable in the array `vars` with respect to the current incumbent solution, and it puts those values into the corresponding array `vals`. If there is no incumbent, this method throws an exception.

```
public IloNum getMIPRelativeGap() const
```

This method accesses the current relative objective gap.

For a **minimization** problem, this value is computed by

```
(bestinteger - bestobjective) / (1e-10 + |bestobjective|)
```

where `bestinteger` is the value returned by `IloCplex::MIPInfoCallbackI::getIncumbentValue` and `bestobjective` is the value returned by `IloCplex::MIPInfoCallbackI::getBestObjValue`. For a **maximization** problem, the value is computed by:

```
(bestobjective - bestinteger) / (1e-10 + |bestobjective|)
```

```
public IloInt getMyThreadNum() const
```

Returns the identifier of the parallel thread being currently executed. This number is between 0 (zero) and the value returned by the method `getUserThreads()-1`.

This method returns a nonzero value **only** for the class `IloCplex::MIPCallbackI` and its subclasses. In other words, this method is valid only for query, diagnostic, and control callbacks. It is **not** valid for informational callbacks.

```
public IloInt getNiterations() const
```

Returns the total number of iterations executed so far during the current optimization to solve the node relaxations.

```
public IloInt getNnodes() const
```

Returns the number of nodes already processed in the current optimization.

```
public IloInt getNremainingNodes() const
```

Returns the number of nodes left to explore in the current optimization.

```
public IloNum getPriority(const IloIntVar sos) const
```

Returns the branch priority used for variable `var` in the current optimization.

```
public IloNum getPriority(const IloNumVar sos) const
```

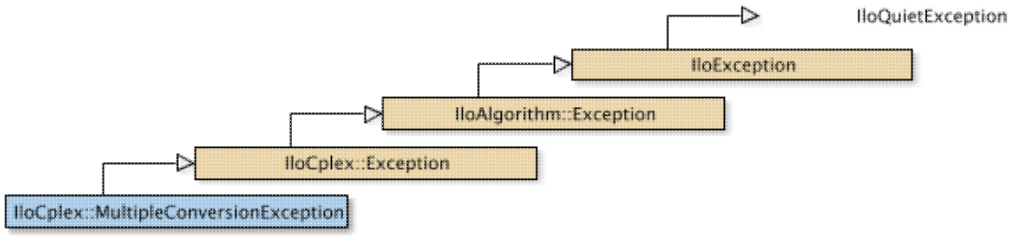
Returns the branch priority used for variable `var` in the current optimization.

```
public IloBool hasIncumbent() const
```

Returns `IloTrue` if an integer feasible solution has been found, or, equivalenty, if an incumbent solution is available at the time the invoking callback is called by an instance of `IloCplex` while solving a MIP.

Class IloCplex::MultipleConversionException

Definition file: ilcplex/ilocplexi.h



An instance of this exception is thrown by `IloCplex` when there is an attempt to convert the type of a variable with more than one `IloConversion` object at a time, while it is being extracted by `IloCplex`.

Method Summary	
public IloConversion	getConversion() const
public const IloNumVarArray	getVariables() const

Inherited Methods from `Exception`
getStatus

Inherited Methods from `IloException`
end, getMessage

Methods

`public IloConversion` **`getConversion`**`() const`

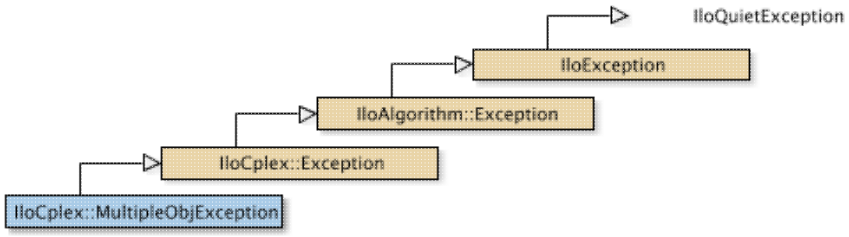
This method returns the offending `IloConversion` object.

`public const IloNumVarArray` **`getVariables`**`() const`

This method returns an array of variables to which too many type conversions have been applied.

Class IloCplex::MultipleObjException

Definition file: ilcplex/ilocplexi.h



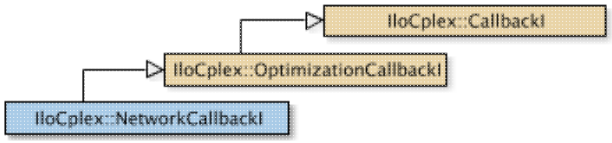
An instance of this exception is thrown by `IloCplex` when there is an attempt to use more than one objective function in a model extracted by `IloCplex`.

Inherited Methods from `Exception`
`getStatus`

Inherited Methods from `IloException`
`end, getMessage`

Class IloCplex::NetworkCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::NetCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` with the network optimizer. The callback is executed each time the network optimizer issues a log file message.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::OptimizationCallbackI, ILONETWORKCALLBACK0

Constructor Summary	
protected	NetworkCallbackI(IloEnv env)

Method Summary	
public IloNum	getInfeasibility() const
public IloInt	getNiterations() const
public IloNum	getObjValue() const
public IloBool	isFeasible() const

Inherited Methods from `OptimizationCallbackI`
getModel, getNcols, getNQCs, getNrows

Inherited Methods from `CallbackI`
abort, duplicateCallback, getEndTime, getEnv, main

Constructors

`protected` **`NetworkCallbackI`**`(IloEnv env)`

This constructor creates a callback for use with the network optimizer.

Methods

`public IloNum` **`getInfeasibility`**`() const`

This method returns the current primal infeasibility measure of the network solution in the instance of `IloCplex` at the time the invoking callback is executed.

```
public IloInt getNiterations() const
```

This method returns the number of network simplex iterations completed so far by an instance of `IloCplex` at the invoking callback is executed.

```
public IloNum getObjValue() const
```

This method returns the current objective value of the network solution in the instance of `IloCplex` at the time the invoking callback is executed.

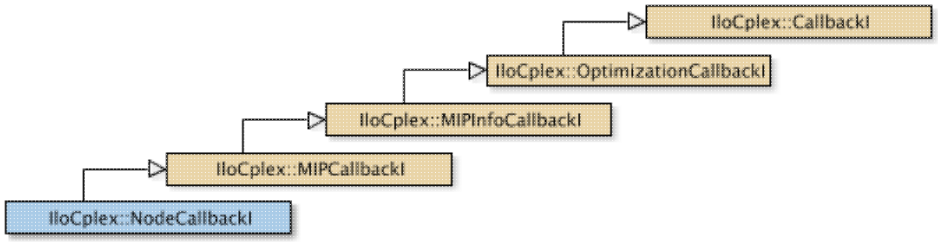
If you need the object representing the objective itself, consider the method `IloCplex::getObjective` instead.

```
public IloBool isFeasible() const
```

This method returns `IloTrue` if the current network solution is primal feasible.

Class IloCplex::NodeCallbackI

Definition file: ilcplex/ilocplexi.h



Note

This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::NodeCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). The methods of this class enable you (from a user-derived callback class) to query the instance of `IloCplex` about the next node that it plans to select in the branch-and-cut search, and optionally to override this selection by specifying a different node.

When an instance of this callback executes, the invoking instance of `IloCplex` still has `n = getNremainingNodes` (inherited from `IloCplex::MIPCallbackI`) nodes left to process. These remaining nodes are numbered from 0 (zero) to `(n - 1)`. For that reason, the same node may have a different number each time an instance of `NodeCallbackI` is called. To identify a node uniquely, an instance of `IloCplex` also assigns a unique `NodeId` to each node. That unique identifier remains unchanged throughout the search. The method `getNodeId(int i)` allows you to access the `NodeId` for each of the remaining nodes (0 to n-1). Such a query allows you to associate data with individual nodes.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI, IloCplex::OptimizationCallbackI, ILONODECALLBACK0

Constructor Summary	
`protected`	`NodeCallbackI(IloEnv env)`

Method Summary	
`public IloNumVar`	`getBranchVar(NodeId nodeid) const`
`public IloNumVar`	`getBranchVar(int node) const`
`public IloInt`	`getDepth(NodeId nodeid) const`
`public IloInt`	`getDepth(int node) const`
`public IloNum`	`getEstimatedObjValue(NodeId nodeid) const`
`public IloNum`	`getEstimatedObjValue(int node) const`
`public IloNum`	`getInfeasibilitySum(NodeId nodeid) const`

public IloNum	getInfeasibilitySum(int node) const
public IloInt	getNinfeasibilities(NodeId nodeid) const
public IloInt	getNinfeasibilities(int node) const
public NodeData *	getNodeData(NodeId nodeid) const
public NodeData *	getNodeData(int node) const
public NodeId	getNodeId(int node) const
public IloInt	getNodeNumber(NodeId nodeid) const
public IloNum	getObjValue(NodeId nodeid) const
public IloNum	getObjValue(int node) const
public void	selectNode(NodeId nodeid)
public void	selectNode(int node)

Inherited Methods from **MIPCallbackI**
getNcliques, getNcovers, getNcuts, getNdisjunctiveCuts, getNflowCovers, getNflowPaths, getNfractionalCuts, getNGUBcovers, getNimpliedBounds, getNMIRs, getNzeroHalfCuts, getObjCoef, getObjCoef, getObjCoefs, getObjCoefs, getUserThreads

Inherited Methods from **MIPInfoCallbackI**
getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent

Inherited Methods from **OptimizationCallbackI**
getModel, getNcols, getNQCs, getNrows

Inherited Methods from **CallbackI**
abort, duplicateCallback, getEndTime, getEnv, main

Constructors

protected **NodeCallbackI**(IloEnv env)

This constructor creates a callback for use in an application with user-defined node selection inquiry during branch-and-cut searches.

Methods

public IloNumVar **getBranchVar**(NodeId nodeid) const

This method returns the variable that was branched on last when CPLEX created the node with the identifier `nodeid`. If that node has been created by branching on a constraint or on multiple variables, 0 (zero) will be returned.

public IloNumVar **getBranchVar**(int node) const

Returns the variable that was branched on last when creating the node specified by the index number `node`. If that node has been created by branching on a constraint or on multiple variables, 0 (zero) will be returned.

```
public IloInt getDepth(NodeId nodeid) const
```

This method returns the depth of the node in the search tree for the node with the identifier `nodeid`. The root node has depth 0 (zero). The depth of other nodes is defined recursively as the depth of their parent node plus one. In other words, the depth of a node is its distance in terms of the number of branches from the root.

```
public IloInt getDepth(int node) const
```

This method returns the depth of the node in the search tree. The root node has depth 0 (zero). The depth of other nodes is defined recursively as the depth of their parent node plus one. In other words, the depth of a node is its distance in terms of the number of branches from the root.

```
public IloNum getEstimatedObjValue(NodeId nodeid) const
```

This method returns the estimated objective value of the node with the identifier `node`.

```
public IloNum getEstimatedObjValue(int node) const
```

This method returns the estimated objective value of the node specified by the index number `node`.

```
public IloNum getInfeasibilitySum(NodeId nodeid) const
```

This method returns the sum of infeasibility measures at the node with the identifier `nodeid`.

```
public IloNum getInfeasibilitySum(int node) const
```

This method returns the sum of infeasibility measures at the node specified by the index number `node`.

```
public IloInt getNinfeasibilities(NodeId nodeid) const
```

This method returns the number of infeasibilities at the node with the identifier `nodeid`.

```
public IloInt getNinfeasibilities(int node) const
```

This method returns the number of infeasibilities at the node specified by the index number `node`.

```
public NodeData * getNodeData(NodeId nodeid) const
```

This method retrieves the `NodeData` object that may have previously been assigned by the user to the node with the identfier `nodeid` with one of the methods `IloCplex::BranchCallbackI::makeBranch`. If no data object has been assigned to the that node, 0 (zero) will be returned.

```
public NodeData * getNodeData(int node) const
```

This method retrieves the `NodeData` object that may have previously been assigned to the node with index `node` by the user with the method `IloCplex::BranchCallbackI::makeBranch`. If no data object has been assigned to the specified node, 0 (zero) will be returned.

```
public NodeId getNodeId(int node) const
```

This method returns the node identifier of the node specified by the index number `node`. During branch and cut, an instance of `IloCplex` assigns node identifiers sequentially from 0 (zero) to `(getNodes - 1)` as it creates nodes. Within a search, these node identifiers are unique throughout the duration of that search. However, at any point, the remaining nodes, (that is, the nodes that have not yet been processed) are stored in an array in an arbitrary order. This method returns the identifier of the node stored at position `node` in that array.

```
public IloInt getNodeNumber(NodeId nodeid) const
```

Returns the current index number of the node specified by the node identifier `nodeid`.

```
public IloNum getObjValue(NodeId nodeid) const
```

This method returns the objective value of the node with the identifier `node`.

If you need the object representing the objective itself, consider the method `IloCplex::getObjective` instead.

```
public IloNum getObjValue(int node) const
```

This method returns the objective value of the node specified by the index number `node`.

If you need the object representing the objective itself, consider the method `IloCplex::getObjective` instead.

```
public void selectNode(NodeId nodeid)
```

This method selects the node with the identifier `nodeid` and sets it as the next node to process in the branch-and-cut search. The invoking instance of `IloCplex` uses the specified node as the next node to process.

```
public void selectNode(int node)
```

This method selects the node specified by its index number `node` and sets it as the next node to process in the branch & cut search. The parameter `node` must be an integer between 0 (zero) and `(getNremainingNodes - 1)`.

The invoking instance of `IloCplex` uses the specified node as the next node to process.

Class MIPCallbackI::NodeData

Definition file: ilcplex/ilocplexi.h

Objects of (a subclass of) this class can be attached to created nodes in a branch callback with one of the `IloCplex::BranchCallbackI::makeBranch` methods. This allows the user to associate arbitrary data with the nodes. The destructor must be implemented to delete all such data. It will typically be called by `IloCplex` when a node is discarded, either because it has been processed or because it is pruned.

See Also: IloCplex::MIPCallbackI, ILOBRANCHCALLBACK0

Constructor and Destructor Summary	
`public`	`~NodeData()`

Method Summary	
`public virtual IloAny`	`getDataType() const`

Constructors and Destructors

`public` **`~NodeData`**`()`

This is the destructor. It is called by `IloCplex` to delete `NodeData` objects asociated with nodes when they are deleted. By overloading this destructor, you can include objects in your `NodeData` class that need special handling for deletion, other than calling the default destructor.

Methods

`public virtual IloAny` **`getDataType`**`() const`

`IloCplex` does not use this method. It is provided as a convenience for the user to help manage different types of `NodeData` subclasses.

Class IloCplex::NodeEvaluator

Definition file: ilcplex/ilocplexi.h

This class is the handle class for objects of type `IloCplex::NodeEvaluatorI`. Node evaluators can be used to control the node selection strategy during goal-controlled search. That is, node evaluators control the order in which nodes are processed during branch-and-cut search using `IloCplex` goals. Such objects allow you to control the node-selection scheme.

`IloCplex::NodeEvaluatorI` objects are reference-counted. In other words, every instance of `IloCplex::NodeEvaluatorI` keeps track of how many handle objects refer to it. When this number drops to 0 (zero), the `IloCplex::NodeEvaluatorI` object is automatically deleted. As a consequence, whenever you deal with node evaluators, you must maintain a handle object rather than just a pointer to the implementation object. Otherwise, you risk ending up with a pointer to an implementation object that has already been deleted.

See Also: IloCplex, IloCplex::NodeEvaluatorI

Constructor and Destructor Summary	
public	NodeEvaluator()
public	NodeEvaluator(IloCplex::NodeEvaluatorI * impl)
public	NodeEvaluator(const NodeEvaluator & eval)
public	~NodeEvaluator()

Method Summary	
public IloCplex::NodeEvaluatorI *	getImpl() const
public NodeEvaluator	operator=(const NodeEvaluator & eval)

Constructors and Destructors

`public` **`NodeEvaluator`**`()`

The empty constructor creates a new evaluator containing no pointers to an implementation object.

`public` **`NodeEvaluator`**`(IloCplex::NodeEvaluatorI * impl)`

This constructor creates a new evaluator with a pointer to an implementation. It increases the reference count of `impl` by one.

`public` **`NodeEvaluator`**`(const NodeEvaluator & eval)`

This copy constructor increments the reference count of the implementation object referenced by `eval` by one.

`public` **`~NodeEvaluator`**`()`

The destructor decreases the reference count of the implementation object by one. If this reduces the reference count to 0 (zero), the implementation object is automatically deleted.

Methods

```
public IloCplex::NodeEvaluatorI * getImpl() const
```

Queries the implementation object.

```
public NodeEvaluator operator=(const NodeEvaluator & eval)
```

The assignment operator increases the reference count of the implementation object of `eval`. If the invoking handle referred to an implementation object before the assignment operation, its reference count is decreased. If this decrement reduces the reference count to 0 (zero), the implementation object is deleted.

Class IloCplex::NodeEvaluatorI

Definition file: ilcplex/ilocplexi.h

`IloCplex::NodeEvaluatorI` is the base class for implementing node evaluators. Node evaluators allow you to control the node selection strategy for a subtree by assigning values to the nodes. By default, `IloCplex` selects the node with the lowest value when choosing the next node to process during branch-and-cut search. This behavior can be altered by overwriting method `subsume`.

To implement your own node evaluator, you need to create a subclass of `IloCplex::NodeEvaluatorI` and implement methods `evaluate` and `duplicateEvaluator`. The method `evaluate` must be implemented to compute and return a value for a given node. The protected methods of class `IloCplex::NodeEvaluatorI` can be called to query information about the node in order to compute this value. Each node is evaluated only once, after which the value is attached to the node until the node is processed or pruned.

The `duplicateEvaluator` method is called by `IloCplex` when a copy of the evaluator must be created for use in parallel branch-and-cut search. Thus the implementation must simply create and return a copy of the evaluator itself—calling the copy constructor will work in most circumstances.

Node evaluators are applied to a search defined by a goal with the method `IloCplex::Apply`. The node selection strategy will be applied only to the subtree defined by the goal passed to `Apply`. Using `IloCplex::Apply`, you can assign different node selection strategies to different subtrees. You can also assign multiple node selection strategies to subtrees. In this case, node selection strategies applied first have precedence over those assigned later.

If no node evaluators are added, `IloCplex` uses the node selection strategy as controlled by the `NodeSel` parameter.

See Also: IloCplex, IloCplex::NodeEvaluator

Constructor and Destructor Summary	
public	NodeEvaluatorI()
public	~NodeEvaluatorI()

Method Summary	
public virtual NodeEvaluatorI *	duplicateEvaluator()
public virtual IloNum	evaluate() const
protected IloNumVar	getBranchVar() const
protected IloNum	getDepth() const
protected IloNum	getEstimatedObjValue() const
protected IloNum	getInfeasibilitySum() const
protected IloInt	getNinfeasibilities() const
protected GoalI::NodeId	getNodeId() const
protected IloNum	getObjValue() const
public virtual void	init()
public virtual IloBool	subsume(IloNum evalBest, IloNum evalCurrent) const

Constructors and Destructors

`public` **`NodeEvaluatorI`**`()`

This constructor creates a node selector for use in an application with a user-defined node selection strategy to solve a MIP.

`public` **`~NodeEvaluatorI`**`()`

The virtual destructor allows you to manage your own data inside a node evaluator and delete it when the evaluator is deleted.

Methods

`public virtual NodeEvaluatorI * ` **`duplicateEvaluator`**`()`

This method must be implemented by the user to return a copy of the invoking object. It is called internally to duplicate the current node evaluator for parallel branch-and-cut search. This method is not called for a particular node, so the `get` methods cannot be used.

`public virtual IloNum` **`evaluate`**`() const`

This method must be implemented by the user to return a value for a given node. When this method is called, the node evaluator is initialized to the node for which to compute the value. Information about this node can be obtained by the `get` methods of `IloCplex::NodeEvaluatorI`. Returning `IloInfinity` instructs `IloCplex` to discard the node being evaluated.

`protected IloNumVar` **`getBranchVar`**`() const`

This method returns the variable that `IloCplex` branched on when creating the node being evaluated from its parent. If the node has been generated with a more complex branch, 0 (zero) will be returned instead. This method can be called only from the methods `init` and `evaluate`.

`protected IloNum` **`getDepth`**`() const`

This method returns the depth in the search tree of the node currently being evaluated. The root node is depth 0 (zero); the depth of the current node is its distance from the root, or equivalently, the number of branches taken to get from the root node to the current node. This member function can be called only from the methods `init` and `evaluate`.

`protected IloNum` **`getEstimatedObjValue`**`() const`

This method returns the estimated objective value for the node being evaluated. It can be called only from the methods `init` and `evaluate`.

`protected IloNum` **`getInfeasibilitySum`**`() const`

This method returns the sum of infeasibility measures at the node being evaluated. It can be called only from the methods `init` and `evaluate`.

```
protected IloInt getNinfeasibilities() const
```

This method returns the number of infeasibilities at the node being evaluated. It can be called only from the methods `init` and `evaluate`.

```
protected GoalI::NodeId getNodeId() const
```

This method returns the node identifier of the node being evaluated. It can be called only from the methods `init` and `evaluate`.

```
protected IloNum getObjValue() const
```

This method returns the objective value of the node being evaluated. It can be called only from the methods `init` and `evaluate`.

If you need the object representing the objective itself, consider the method `IloCplex::getObjective` instead.

```
public virtual void init()
```

This method is called by `IloCplex` immediately before the first time `evaluate` is called for a node, allowing you to initialize the evaluator based on that node. Information about the current node can be queried by calling the `get` methods of `IloCplex::NodeEvaluatorI`.

```
public virtual IloBool subsume(IloNum evalBest, IloNum evalCurrent) const
```

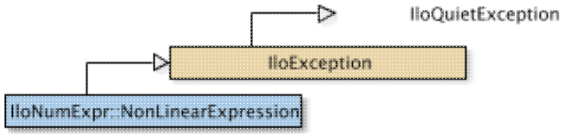
`IloCplex` maintains a candidate node for selection as the next node to process. When choosing the next node, it compares the candidate to all other nodes. If a given node and the candidate node are governed by the same evaluator, `IloCplex` calls `subsume` to determine whether the node should become the new candidate. The arguments passed to the `subsume` call are:

- the value previously assigned by the method `evaluate` to the candidate node as parameter `evalBest`, and
- the value previously assigned by the method `evaluate` to the node under investigation as parameter `evalCurrent`.

By default, this method returns `IloTrue` if `evalCurrent>evalBest`. Overwriting this function allows you to change this selection scheme.

Class IloNumExpr::NonLinearExpression

Definition file: ilconcert/iloexpression.h



The class of exceptions thrown if a numeric constant of a nonlinear expression is set or queried.

Method Summary	
`public const IloNumExprArg`	`getExpression() const`

Inherited Methods from `IloException`
`end, getMessage`

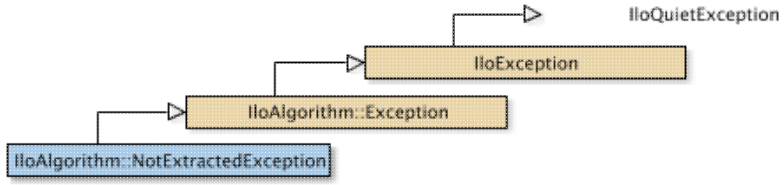
Methods

`public const IloNumExprArg` **`getExpression`**`() const`

The member function `getExpression` returns the expression involved in the exception.

Class IloAlgorithm::NotExtractedException

Definition file: ilconcert/iloalg.h



The class of exceptions thrown if an extractable object has no value in the current solution of an algorithm.
If an expression, numeric variable, objective, or array of extractable objects has no value in the current solution of an algorithm, this exception is thrown.

Constructor Summary	
public	NotExtractedException(const IloAlgorithmI *, const IloExtractable)

Method Summary	
public const IloAlgorithmI *	getAlgorithm() const
public const IloExtractable &	getExtractable()

Inherited Methods from `IloException`
end, getMessage

Constructors

`public` **`NotExtractedException`**`(const IloAlgorithmI *, const IloExtractable)`

The constructor `NotExtractedException` creates an exception thrown from the algorithm object `alg` for the extractable object `extr`.

Methods

`public const IloAlgorithmI * ` **`getAlgorithm`**`() const`

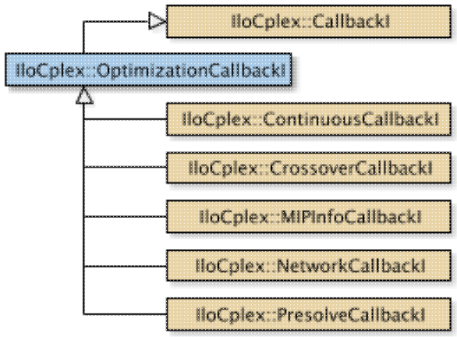
The member function `getAlgorithm` returns the algorithm from which the exception was thrown.

`public const IloExtractable & ` **`getExtractable`**`()`

The member function `getExtractable` returns the extractable object that triggered the exception.

Class IloCplex::OptimizationCallbackI

Definition file: ilcplex/ilocplexi.h



This is the abstract base class for user-written callback classes called by optimization methods. It provides their common application programming interface (API).

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI

Method Summary	
public IloModel	getModel() const
public IloInt	getNcols() const
public IloInt	getNQCs() const
public IloInt	getNrows() const

Inherited Methods from `CallbackI`
abort, duplicateCallback, getEndTime, getEnv, main

Methods

`public IloModel` **`getModel`**`() const`

This method returns the model currently extracted for the instance of `IloCplex` where the invoking callback executed.

`public IloInt` **`getNcols`**`() const`

This method returns the number of columns in the model currently being optimized.

`public IloInt` **`getNQCs`**`() const`

This method returns the number of quadratic constraints in the model currently being optimized.

`public IloInt` **`getNrows`**`() const`

This method returns the number of rows in the model currently being optimized.

Class IloCplex::ParameterSet

Definition file: ilcplex/ilocplexi.h

A parameter set for `IloCplex`, this class allows you to store and restore parameters that are not at their default value.

You can create empty `IloCplex::ParameterSet` objects with the constructor and then modify them. Alternatively, you can create such objects with the method `IloCplex::getParameterSet`.

A parameter set can be applied to an instance of `IloCplex` by means of the method `IloCplex::setParameterSet(set)`.

See Also: IloCplex::getParameterSet, IloCplex::setParameterSet

Method Summary	
public void	clear()
public void	end()
public IloNum	getParam(IloCplex::NumParam which) const
public IloBool	getParam(IloCplex::BoolParam which) const
public char *	getParam(IloCplex::StringParam which) const
public IloInt	getParam(IloCplex::IntParam which) const
public void	setParam(IloCplex::NumParam which, IloNum val)
public void	setParam(IloCplex::BoolParam which, IloBool val)
public void	setParam(IloCplex::StringParam which, char * val)
public void	setParam(IloCplex::IntParam which, IloInt val)

Inner Class
ParameterSet::Iterator

Methods

`public void` **`clear`**`()`

Clears the parameter set.

`public void` **`end`**`()`

Ends the parameter set.

`public IloNum` **`getParam`**`(IloCplex::NumParam which) const`

Returns the current value of a numeric parameter.

If the method fails, an exception of type `IloException`, or one of its derived classes, is thrown.

Parameters:

which The identifier of the num parameter to be queried.
Returns:

The current value of the num parameter.

```
public IloBool getParam(IloCplex::BoolParam which) const
```

Returns the current value of a Boolean parameter.

If the method fails, an exception of type `IloException`, or one of its derived classes, is thrown.

Parameters:

which The identifier of the Boolean parameter to be queried.
Returns:

The current value of the Boolean parameter.

```
public char * getParam(IloCplex::StringParam which) const
```

Returns the current value of a string parameter.

If the method fails, an exception of type `IloException`, or one of its derived classes, is thrown.

Parameters:

which The identifier of the string parameter to be queried.
Returns:

The current value of the string parameter.

```
public IloInt getParam(IloCplex::IntParam which) const
```

Returns the current value of an integer parameter.

If the method fails, an exception of type `IloException`, or one of its derived classes, is thrown.

Parameters:

which The identifier of the integer parameter to be queried.
Returns:

The current value of the integer parameter.

```
public void setParam(IloCplex::NumParam which, IloNum val)
```

Sets a numeric parameter to the value `val`.

Parameters:

which The identifier of the num parameter to be set.
val The new value for the num parameter

```
public void setParam(IloCplex::BoolParam which, IloBool val)
```

Sets a Boolean parameter to the value `val`.

Parameters:

which The identifier of the Boolean parameter to be set.
val The new value for the Boolean parameter.

```
public void setParam(IloCplex::StringParam which, char * val)
```

Sets a string parameter to the value `val`.

Parameters:

which The identifier of the string parameter to set.
val The new value for the string parameter.

```
public void setParam(IloCplex::IntParam which, IloInt val)
```

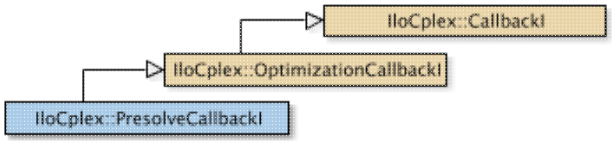
Sets an integer parameter to the value `val`.

Parameters:

which The identifier of the parameter to set.
val The new value for the integer parameter.

Class IloCplex::PresolveCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of a class derived from `PresolveCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex`. The callback is called periodically during presolve. This class enables you to access information about the effects of presolve on the model extracted for the instance of `IloCplex`. For example, there are member functions that return the number of rows or columns removed from the model, the number of variables that have been aggregated, and the number of coefficients that have changed as a result of presolve.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::OptimizationCallbackI, ILOPRESOLVECALLBACK0

Constructor Summary	
protected	PresolveCallbackI(IloEnv env)

Method Summary	
public IloInt	getNaggregations() const
public IloInt	getNmodifiedCoeffs() const
public IloInt	getNremovedCols() const
public IloInt	getNremovedRows() const

Inherited Methods from `OptimizationCallbackI`
getModel, getNcols, getNQCs, getNrows

Inherited Methods from `CallbackI`
abort, duplicateCallback, getEndTime, getEnv, main

Constructors

protected **PresolveCallbackI**(IloEnv env)

This constructor creates a callback for use in presolve.

Methods

public IloInt **getNaggregations**() const

This method returns the number of aggregations performed by presolve at the time the callback is executeed.

```
public IloInt getNmodifiedCoeffs() const
```

This method returns the number of coefficients modified by presolve at the time the callback is executeed.

```
public IloInt getNremovedCols() const
```

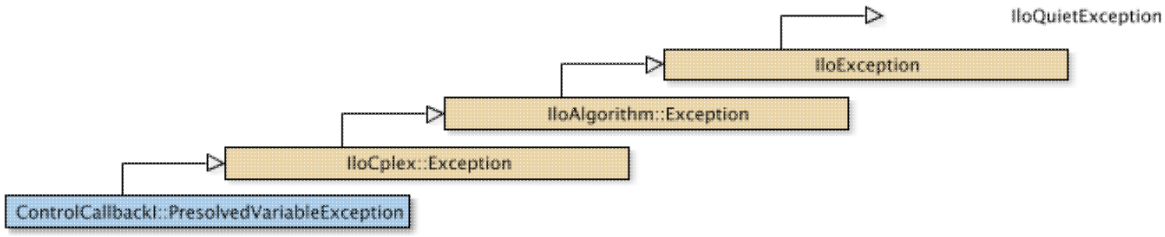
This method returns the number of columns removed by presolve at the time the callback is executeed.

```
public IloInt getNremovedRows() const
```

This method returns the number of rows removed by presolve at the time the callback is executeed.

Class ControlCallbackI::PresolvedVariableException

Definition file: ilcplex/ilocplexi.h



Some operations within a control callback, in particular setting bounds in a heuristic callback, are not possible to do on variables that have been taken out by presolve. An exception of this type is thrown, if such an operation is attempted. Possible ways to avoid this exception are to avoid the operation on presolved variables, to use the method `IloCplex::protectVariables` to protect the variables from being taken out by presolve, or to turn off presolve.

Method Summary	
`public void`	`end()`
`public void`	`getPresolvedVariables(IloNumVarArray vars) const`

Inherited Methods from `Exception`
`getStatus`

Inherited Methods from `IloException`
`end, getMessage`

Methods

`public void` **`end`**`()`

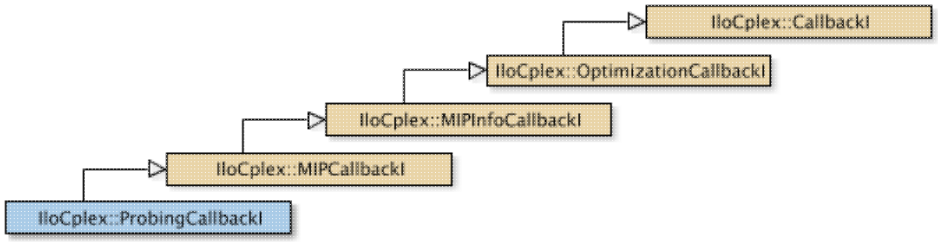
This method must be called to free up the memory used by the exception when the invoking exception object is no longer needed.

`public void` **`getPresolvedVariables`**`(IloNumVarArray vars) const`

This methods copies the variables that caused the invoking exception into the array `vars`.

Class IloCplex::ProbingCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::ProbingCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). This class offers a method to check on the progress of a probing operation.

This class is **not** compatible with dynamic search. If you are looking for support for a user-written callback compatible with dynamic search, consider instead the class `IloCplex::ProbingInfoCallbackI`.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI, IloCplex::OptimizationCallbackI, ILOPROBINGCALLBACK0

Constructor Summary	
protected	ProbingCallbackI(IloEnv env)

Method Summary	
public IloInt	getPhase() const
public IloNum	getProgress() const

Inherited Methods from MIPCallbackI
getNcliques, getNcovers, getNcuts, getNdisjunctiveCuts, getNflowCovers, getNflowPaths, getNfractionalCuts, getNGUBcovers, getNimpliedBounds, getNMIRs, getNzeroHalfCuts, getObjCoef, getObjCoef, getObjCoefs, getObjCoefs, getUserThreads

Inherited Methods from MIPInfoCallbackI
getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent

Inherited Methods from OptimizationCallbackI
getModel, getNcols, getNQCs, getNrows

Inherited Methods from CallbackI
abort, duplicateCallback, getEndTime, getEnv, main

Constructors

`protected` **`ProbingCallbackI`**`(IloEnv env)`

This constructor creates a callback for use in an application when probing.

Methods

`public IloInt` **`getPhase`**`() const`

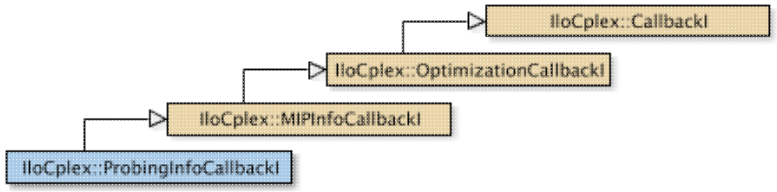
This method returns the current phase of probing.

`public IloNum` **`getProgress`**`() const`

This method returns the fraction of completion of the current probing phase.

Class IloCplex::ProbingInfoCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::ProbingInfoCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). This class offers a method to check on the progress of a probing operation.

User-written callbacks of this class are compatible with MIP dynamic search.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPInfoCallbackI, IloCplex::OptimizationCallbackI, ILOPROBINGINFOCALLBACK0

Constructor Summary	
protected	ProbingInfoCallbackI(IloEnv env)

Method Summary	
public IloInt	getPhase() const
public IloNum	getProgress() const

Inherited Methods from MIPInfoCallbackI
getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent

Inherited Methods from OptimizationCallbackI
getModel, getNcols, getNQCs, getNrows

Inherited Methods from CallbackI
abort, duplicateCallback, getEndTime, getEnv, main

Constructors

protected **ProbingInfoCallbackI**(IloEnv env)

This constructor creates a callback for use in an application when probing.

Methods

```
public IloInt getPhase() const
```

This method returns the current phase of probing.

```
public IloNum getProgress() const
```

This method returns the fraction of completion of the current probing phase.

Class IloCplex::SearchLimit

Definition file: ilcplex/ilocplexi.h

Search limits can be used to impose limits on the exploration of certain subtrees during branch-and-cut search. Search limits are implemented in the class `IloCplex::SearchLimitI`. This is the handle class for CPLEX search limits.

The search limit objects are reference-counted. This reference counting means an instance of `IloCplex::SearchLimitI` keeps track of how many handle objects refer to it. If this number drops to 0, the `IloCplex::SearchLimitI` object is automatically deleted. As a consequence, whenever you deal with a search limit, you must maintain a handle object rather then only a pointer to the implementation object. Otherwise, you risk ending up with a pointer to an implementation object that has already been deleted.

See Also: IloCplex, IloCplex::SearchLimitI

Constructor and Destructor Summary	
public	`SearchLimit()`
public	`SearchLimit(IloCplex::SearchLimitI * impl)`
public	`SearchLimit(const SearchLimit & limit)`
public	`~SearchLimit()`

Method Summary	
public IloCplex::SearchLimitI *	`getImpl() const`
public SearchLimit	`operator=(const SearchLimit & limit)`

Constructors and Destructors

`public` **`SearchLimit`**`()`

The default constructor creates a new search limit with 0 implementation object pointer.

`public` **`SearchLimit`**`(IloCplex::SearchLimitI * impl)`

This constructor creates a new search limit with a pointer to an implementation. It increases the reference count of `impl` by one.

`public` **`SearchLimit`**`(const SearchLimit & limit)`

This copy constructor increments the reference count of the implementation object referenced by `limit` by one.

`public` **`~SearchLimit`**`()`

The destructor decreases the reference count of `impl` by one. If this reduces the reference count to 0, the implementation object is automatically deleted.

Methods

```
public IloCplex::SearchLimitI * getImpl() const
```

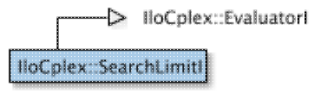
Queries the implementation object of the invoking search limit.

```
public SearchLimit operator=(const SearchLimit & limit)
```

The assignment operator increases the reference count of the implementation object of `limit`. If the invoking handle referred to an implementation object before the assignment operation, its reference count is decreased. If this reduces the reference count to 0, the implementation object is deleted.

Class IloCplex::SearchLimitI

Definition file: ilcplex/ilocplexi.h



`IloCplex::SearchLimitI` is the base class for implementing user-defined search limits. To do so, you must subclass `IloCplex::SearchLimitI` and implement methods `check` and `duplicateLimit`. You may optionally implement method `init`. The method `check` must return `IloTrue` when the limit is reached and `IloFalse` otherwise. The method `duplicateLimit` must return a copy of the invoking object to be used in parallel search.

Whenever method `check` is called by `IloCplex`, the search limit object is first initialized to a node, referred to as the current node. Information about the current node can be queried by calling the `get` methods of class `IloCplex::SearchLimitI`.

Search limits are applied to subtrees defined by goals with the method `IloCplex::LimitSearch`. For example:

```
IloGoal limitGoal = IloCplex::LimitSearch(cplex, goal1, limit);
```

creates a goal `limitGoal` which branches as specified by `goal1` until the limit specified by `limit` is reached. Only the nodes created by `goal1` (or any of the goals created by it later) are subjected to the search limit. For example, if you created two branches with the goal

```
OrGoal(limitGoal, goal2);
```

only the subtree defined by `goal1` is subject to the search limit `limit`; the subtree defined by `goal2` is not.

The ability to specify search limits for subtrees means that it is possible for certain branches to be subject to more than one search limit. Nodes with multiple search limits attached to them are processed only if none of the search limits has been reached, or, in other words, if all the search limits return `IloFalse` when method `check` is called by `IloCplex`.

Each time CPLEX uses a search limit, it is duplicated first. If you use the same instance of your limit in different branches, it will be duplicated first, the copy will be passed to the corresponding node, and `init` method will be called on the copy.

See Also: IloCplex, IloCplex::SearchLimit

Constructor and Destructor Summary	
public	SearchLimitI()
public	~SearchLimitI()

Method Summary	
public virtual IloBool	check()
public virtual SearchLimitI *	duplicateLimit()
public virtual void	init()

Constructors and Destructors

`public` **`SearchLimitI`**`()`

The default constructor creates a new instance of `SearchLimitI`.

`public` **`~SearchLimitI`**`()`

The virtual destructor allows you to manage you own data inside a search limit object and to delete the data when the limit is deleted.

Methods

`public virtual IloBool` **`check`**`()`

This method is called for every node subjected to the invoking search limit before evaluating the node. If it returns `IloTrue`, the node is pruned, or, equivalently, the search below that node is discontinued. Thus, users implementing search limits must implement this method to return `IloTrue` if the search limit has been reached and `IloFalse` otherwise.

`public virtual SearchLimitI * ` **`duplicateLimit`**`()`

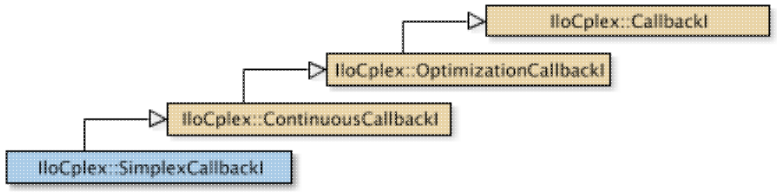
This method is called internally to duplicate the current search limit. Users must implement it in a subclass to return a copy of the invoking object.

`public virtual void` **`init`**`()`

This method is called by `IloCplex` right before the first time `check` is called for a node and allows you to initialize the limit based on that node.

Class IloCplex::SimplexCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of the class `IloCplex::SimplexCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a problem by means of the simplex optimizer. For more information on the simplex optimizer, see the *CPLEX User's Manual*. `IloCplex` calls the user-written callback after each iteration during optimization with the simplex algorithm.

The constructor of this class is protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::ContinuousCallbackI, IloCplex::OptimizationCallbackI, ILOSIMPLEXCALLBACK0

Constructor Summary	
protected	SimplexCallbackI(IloEnv env)

Inherited Methods from ContinuousCallbackI
getDualInfeasibility, getInfeasibility, getNiterations, getObjValue, isDualFeasible, isFeasible

Inherited Methods from OptimizationCallbackI
getModel, getNcols, getNQCs, getNrows

Inherited Methods from CallbackI
abort, duplicateCallback, getEndTime, getEnv, main

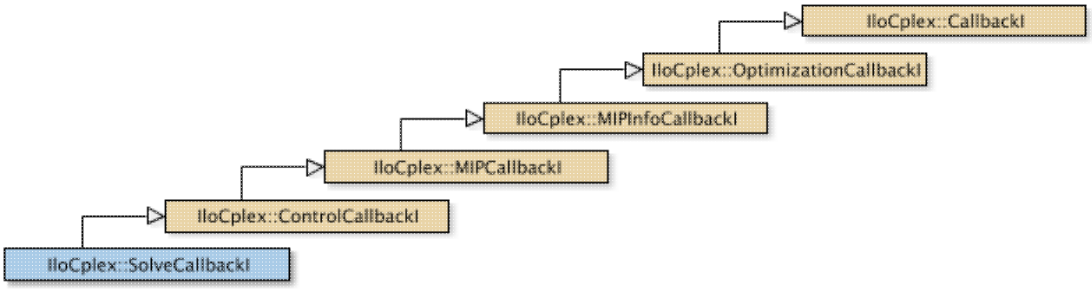
Constructors

protected **SimplexCallbackI**(IloEnv env)

This constructor creates a callback for use in an application of the simplex optimizer.

Class IloCplex::SolveCallbackI

Definition file: ilcplex/ilocplexi.h



Note

This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::SolveCallbackI` can be used to solve subproblems (for example, node and heuristic subproblems) during branch-and-cut search. It allows you to set a starting point for the solve or to select the algorithm on a per-node basis.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::ControlCallbackI, IloCplex::OptimizationCallbackI, ILOSOLVECALLBACK0

Constructor Summary	
protected	SolveCallbackI(IloEnv env)

Method Summary	
public IloCplex::CplexStatus	getCplexStatus() const
public IloAlgorithm::Status	getStatus() const
public IloBool	isDualFeasible() const
public IloBool	isPrimalFeasible() const
public void	setVectors(const IloNumArray x, const IloIntVarArray var, const IloNumArray pi, const IloRangeArray rng)
public void	setVectors(const IloNumArray x, const IloNumVarArray var, const IloNumArray pi, const IloRangeArray rng)
public IloBool	solve(IloCplex::Algorithm alg=Dual)
public void	useSolution()

Inherited Methods from `ControlCallbackI`
getDownPseudoCost, getDownPseudoCost, getFeasibilities, getFeasibilities, getFeasibility, getFeasibility, getFeasibility, getFeasibility, getLB, getLB,

getLBs, getLBs, getNodeData, getObjValue, getSlack, getSlacks, getUB, getUB, getUBs, getUBs, getUpPseudoCost, getUpPseudoCost, getValue, getValue, getValue, getValues, getValues, isSOSFeasible, isSOSFeasible

Inherited Methods from **MIPCallbackI**
getNcliques, getNcovers, getNcuts, getNdisjunctiveCuts, getNflowCovers, getNflowPaths, getNfractionalCuts, getNGUBcovers, getNimpliedBounds, getNMIRs, getNzeroHalfCuts, getObjCoef, getObjCoef, getObjCoefs, getObjCoefs, getUserThreads

Inherited Methods from **MIPInfoCallbackI**
getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent

Inherited Methods from **OptimizationCallbackI**
getModel, getNcols, getNQCs, getNrows

Inherited Methods from **CallbackI**
abort, duplicateCallback, getEndTime, getEnv, main

Constructors

protected **SolveCallbackI**(IloEnv env)

This constructor creates a callback for use in an application for solving the node LPs during branch-and-cut searches.

Methods

public IloCplex::CplexStatus **getCplexStatus**() const

This method returns the CPLEX status of the instance of IloCplex at the current node (that is, the state of the optimizer at the node) during the last call to SolveCallbackI::solve (which may have been called directly in the callback or by IloCplex when processing the node).

The enumeration IloCplex::CplexStatus lists the possible status values.

public IloAlgorithm::Status **getStatus**() const

This method returns the status of the solution found by the instance of IloCplex at the current node during the last call to SolveCallbackI::solve (which may have been called directly in the callback or by IloCplex when processing the node).

The enumeration IloAlgorithm::Status lists the possible status values.

public IloBool **isDualFeasible**() const

This method returns `IloTrue` if the solution provided by the last `solve` call is dual feasible. Note that an `IloFalse` return value does not necessarily mean that the solution is not dual feasible. It simply means that the relevant algorithm was not able to conclude it was dual feasible when it terminated.

```
public IloBool isPrimalFeasible() const
```

This method returns `IloTrue` if the solution provided by the last `solve` call is primal feasible. Note that an `IloFalse` return value does not necessarily mean that the solution is not primal feasible. It simply means that the relevant algorithm was not able to conclude it was primal feasible when it terminated.

```
public void setVectors(const IloNumArray x, const IloIntVarArray var, const
IloNumArray pi, const IloRangeArray rng)
```

This method allows a user to specify a starting point for the following invocation of the `solve` method in a solve callback. Zero can be passed for any of the parameters. However, if `x` is not zero, then `var` must not be zero either. Similarly, if `pi` is not zero, then `rng` must not be zero either.

For all variables in `var`, `x[i]` specifies the starting value for the variable `var[i]`. Similarly, for all ranged constraints specified in `rng`, `pi[i]` specifies the starting dual value for `rng[i]`.

This information is exploited at the next call to `solve`, to construct a starting point for the algorithm.

```
public void setVectors(const IloNumArray x, const IloNumVarArray var, const
IloNumArray pi, const IloRangeArray rng)
```

This method allows a user to specify a starting point for the following invocation of the `solve` method in a solve callback. Zero can be passed for any of the parameters. However, if `x` is not zero, then `var` must not be zero either. Similarly, if `pi` is not zero, then `rng` must not be zero either.

For all variables in `var`, `x[i]` specifies the starting value for the variable `var[i]`. Similarly, for all ranged constraints specified in `rng`, `pi[i]` specifies the starting dual value for `rng[i]`.

This information is exploited at the next call to `solve`, to construct a starting point for the algorithm.

```
public IloBool solve(IloCplex::Algorithm alg=Dual)
```

This method uses the algorithm `alg` to solve the current node LP. See `IloCplex::Algorithm` for a choice of algorithms to use.

```
public void useSolution()
```

A call to this method instructs `IloCplex` to use the solution generated with this callback.

If `useSolution` is not called, `IloCplex` uses the algorithm selected with the parameters `IloCplex::RootAlg` for the solution of the root, or `IloCplex::NodeAlg` to solve the node.

Class IloCsvReader::TableIterator

Definition file: ilconcert/ilocsvreader.h

Table-iterator of csv readers.
`TableIterator` is a nested class of the class `IloCsvReader`. It is to be used only for multitable files.

`IloCsvReader::TableIterator` allows you to step through all the tables of the multitable csv data file on which the csv reader was created.

Constructor and Destructor Summary	
`public`	`TableIterator(IloCsvReader csv)`

Method Summary	
`public IloBool`	`ok() const`
`public IloCsvTableReader`	`operator*() const`
`public TableIterator &`	`operator++()`

Constructors and Destructors

`public` **`TableIterator`**`(IloCsvReader csv)`

This constructor creates an iterator to traverse all the tables in the csv data file on which the csv reader `csv` was created.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one.

It returns `IloFalse` if the iterator reaches the end of the table.

`public IloCsvTableReader` **`operator*`**`() const`

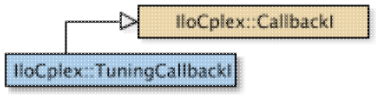
This operator returns the current instance of `IloCsvTable` (representing the current table in the csv file); the one to which the invoking iterator points.

`public TableIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloCsvTableReader` representing the next line in the file.

Class IloCplex::TuningCallbackI

Definition file: ilcplex/ilocplexi.h



An instance of a class derived from `TuningCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex`. The callback is called periodically during tuning. This class enables you to access information on the progress of tuning.

See Also: IloCplex, IloCplex::Callback, IloCplex::CallbackI, ILOTUNINGCALLBACK0

Constructor Summary	
protected	TuningCallbackI(IloEnv env)

Method Summary	
public IloNum	getProgress() const

Inherited Methods from `CallbackI`
abort, duplicateCallback, getEndTime, getEnv, main

Constructors

`protected` **`TuningCallbackI`**`(IloEnv env)`

This constructor creates a callback for use in an application to monitor tuning progress.

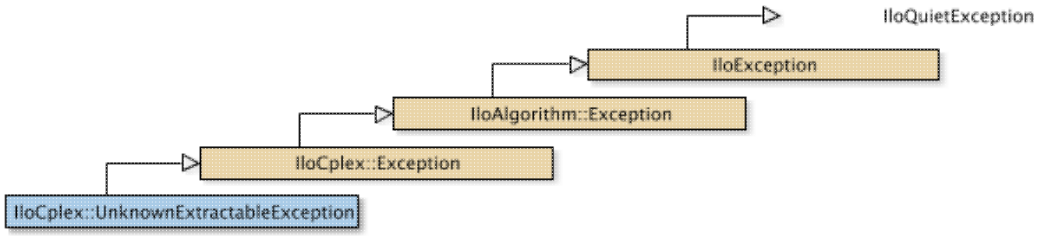
Methods

`public IloNum` **`getProgress`**`() const`

This method returns the fraction of completion of the tuning process.

Class IloCplex::UnknownExtractableException

Definition file: ilcplex/ilocplexi.h



An instance of this exception is thrown by `IloCplex` when an operation is attempted using an extractable that has not been extracted.

Method Summary	
`public IloExtractable`	`getExtractable() const`

Inherited Methods from `Exception`
`getStatus`

Inherited Methods from `IloException`
`end, getMessage`

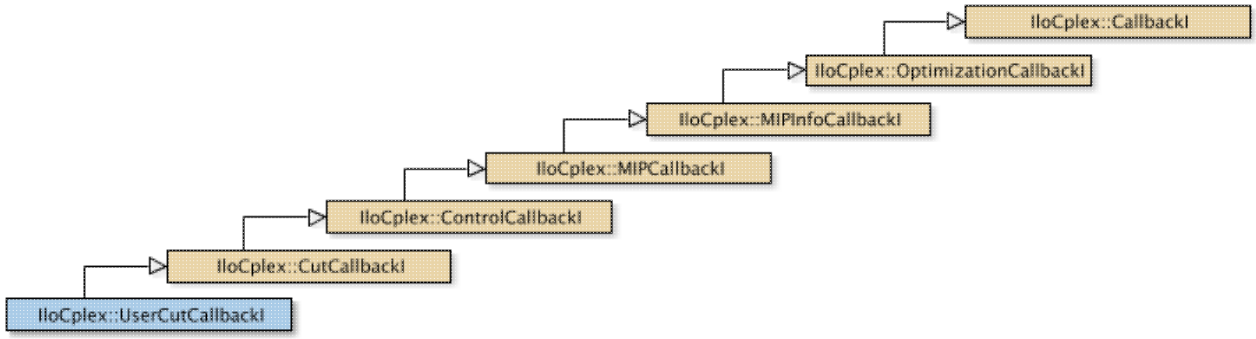
Methods

`public IloExtractable` **`getExtractable`**`() const`

This method returns the offending extractable object.

Class IloCplex::UserCutCallbackI

Definition file: ilcplex/ilocplexi.h



Note

This is an advanced class. Advanced classes typically demand a profound understanding of the algorithms used by CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, the team encourages you to consider carefully whether you can accomplish the same task by means of other classes instead.

An instance of the class `IloCplex::UserCutCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a MIP while generating user cuts to tighten the LP relaxation. `IloCplex` calls the user-written callback after solving each node LP exactly like `IloCplex::CutCallbackI`. It differs from `IloCplex::CutCallbackI` only in that constraints added in a `UserCutCallbackI` must be real cuts in the sense that omitting them does not affect the feasible region of the model under consideration.

Method Summary	
`protected IloConstraint`	`add(IloConstraint con, IloBool purgeable=IloFalse)`

Inherited Methods from `CutCallbackI`
`add, addLocal`

Inherited Methods from `ControlCallbackI`
`getDownPseudoCost, getDownPseudoCost, getFeasibilities, getFeasibilities, getFeasibility, getFeasibility, getFeasibility, getFeasibility, getLB, getLB, getLBs, getLBs, getNodeData, getObjValue, getSlack, getSlacks, getUB, getUB, getUBs, getUBs, getUpPseudoCost, getUpPseudoCost, getValue, getValue, getValue, getValues, getValues, isSOSFeasible, isSOSFeasible`

Inherited Methods from `MIPCallbackI`
`getNcliques, getNcovers, getNcuts, getNdisjunctiveCuts, getNflowCovers, getNflowPaths, getNfractionalCuts, getNGUBcovers, getNimpliedBounds, getNMIRs, getNzeroHalfCuts, getObjCoef, getObjCoef, getObjCoefs, getObjCoefs, getUserThreads`

Inherited Methods from `MIPInfoCallbackI`
`getBestObjValue, getCutoff, getDirection, getDirection, getIncumbentObjValue, getIncumbentSlack, getIncumbentSlacks, getIncumbentValue, getIncumbentValue, getIncumbentValues, getIncumbentValues, getMIPRelativeGap, getMyThreadNum, getNiterations, getNnodes, getNremainingNodes, getPriority, getPriority, hasIncumbent`

Inherited Methods from `OptimizationCallbackI`
`getModel, getNcols, getNQCs, getNrows`

Inherited Methods from `CallbackI`
`abort, duplicateCallback, getEndTime, getEnv, main`

Methods

```
protected IloConstraint add(IloConstraint con, IloBool purgeable=IloFalse)
```

This method adds a cut to the current node LP subproblem for the constraint specified by `con`. This cut must be globally valid. The added cut must be linear. It will not be removed by backtracking. If you designate the cut as *purgeable*, then CPLEX may eliminate it under certain circumstances.

Parameters:

con	The constraint to be added as cut
purgeable	A Boolean value specifying whether CPLEX is allowed to purge the cut during branch and cut if CPLEX deems the cut no longer helpful.

Enumeration BranchType

Definition file: ilcplex/ilocplexi.h

`IloCplex::BranchCallbackI::BranchType` is an enumeration limited in scope to the class `IloCplex::BranchCallbackI`. This enumeration is used by the method `IloCplex::BranchCallbackI::getBranchType` to tell what kind of branch `IloCplex` is about to do:

- `BranchOnVariable` specifies branching on a single variable.
- `BranchOnAny` specifies multiple bound changes and constraints will be used for branching.
- `BranchOnSOS1` specifies branching on an SOS of type 1.
- `BranchOnSOS2` specifies branching on an SOS of type 2.

See Also: IloCplex::BranchCallbackI

Fields:

`BranchOnVariable = CPX_TYPE_VAR`	= CPX_TYPE_VAR
`BranchOnSOS1 = CPX_TYPE_SOS1`	= CPX_TYPE_SOS1
`BranchOnSOS2 = CPX_TYPE_SOS2`	= CPX_TYPE_SOS2
`BranchOnAny = CPX_TYPE_ANY`	= CPX_TYPE_ANY
`UserBranch = CPX_TYPE_USER`	

Enumeration Type

Definition file: ilcplex/ilocplexi.h

This enumeration type is used to identify the type of a callback implementation object referred to by an `IloCplex::Callback` handle.

See Also: IloCplex::Callback

Fields:

```
Presolve = 0
Simplex = 1
Barrier = 2
Crossover = 3
Network = 4
MIP = 5
Probing = 6
FractionalCut = 7
DisjunctiveCut = 8
Branch = 9
Cut = 10
Node = 11
Heuristic = 12
Incumbent = 13
Solve = 14
FlowMIRCut = 15
Continuous = 16
MIPInfo = 17
ProbingInfo = 18
FractionalCutInfo = 19
DisjunctiveCutInfo = 20
FlowMIRCutInfo = 21
Tuning = 22
_Number = 23
```

Enumeration IntegerFeasibility

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::ControlCallbackI::IntegerFeasibility` is an enumeration limited in scope to the class `IloCplex::ControlCallbackI`. This enumeration is used by `IloCplex::ControlCallbackI::getFeasibility` to represent the integer feasibility of a variable or SOS in the current node solution:

- `Feasible` specifies the variable or SOS is integer feasible.
- `ImpliedFeasible` specifies the variable or SOS has been presolved out. It will be feasible when all other integer variables or SOS are integer feasible.
- `Infeasible` specifies the variable or SOS is integer infeasible.

See Also: IloCplex, ControlCallbackI::IntegerFeasibilityArray

Fields:

`ImpliedInfeasible = -1`	
`Feasible = CPX_INTEGER_FEASIBLE`	Not applicable = CPX_INTEGER_FEASIBLE
`Infeasible = CPX_INTEGER_INFEASIBLE`	= CPX_INTEGER_INFEASIBLE
`ImpliedFeasible = CPX_IMPLIED_INTEGER_FEASIBLE`	= CPX_IMPLIED_INTEGER_FEASIBLE

Enumeration BranchType

Definition file: ilcplex/ilocplexi.h

`IloCplex::GoalI::BranchType` is an enumeration limited in scope to the class `IloCplex::GoalI`. This enumeration is used by the method `IloCplex::GoalI::getBranchType` to tell what kind of branch `IloCplex` is about to make:

- `BranchOnVariable` specifies branching on a single variable.
- `BranchOnAny` specifies multiple bound changes and constraints will be used for branching.
- `BranchOnSOS1` specifies branching on an SOS of type 1.
- `BranchOnSOS2` specifies branching on an SOS of type 2.

See Also: IloCplex::GoalI

Fields:

`BranchOnVariable = CPX_TYPE_VAR`	= CPX_TYPE_VAR
`BranchOnSOS1 = CPX_TYPE_SOS1`	= CPX_TYPE_SOS1
`BranchOnSOS2 = CPX_TYPE_SOS2`	= CPX_TYPE_SOS2
`BranchOnAny = CPX_TYPE_ANY`	= CPX_TYPE_ANY
`UserBranch = CPX_TYPE_USER`	

Enumeration IntegerFeasibility

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::GoalI::IntegerFeasibility` is an enumeration limited in scope to the class `IloCplex::GoalI`. This enumeration is used by `IloCplex::GoalI::getFeasibility` to access the integer feasibility of a variable or SOS in the current node solution:

- `Feasible` specifies the variable or SOS is integer feasible.
- `ImpliedFeasible` specifies the variable or SOS has been presolved out. It will be feasible when all other integer variables or SOS are integer feasible.
- `Infeasible` specifies the variable or SOS is integer infeasible.

See Also: IloCplex, GoalI::IntegerFeasibilityArray, ControlCallbackI::IntegerFeasibility

Fields:

`ImpliedInfeasible = -1`	
`Feasible = CPX_INTEGER_FEASIBLE`	Not applicable = CPX_INTEGER_FEASIBLE
`Infeasible = CPX_INTEGER_INFEASIBLE`	= CPX_INTEGER_INFEASIBLE
`ImpliedFeasible = CPX_IMPLIED_INTEGER_FEASIBLE`	= CPX_IMPLIED_INTEGER_FEASIBLE

Enumeration Status

Definition file: ilconcert/iloalg.h

An enumeration for the class `IloAlgorithm`.
`IloAlgorithm` is the base class of algorithms in Concert Technology, and `IloAlgorithm::Status` is an enumeration limited in scope to the class `IloAlgorithm`. The member function `IloAlgorithm::getStatus` returns a status showing information about the current model and the solution.

`Unknown` specifies that the algorithm has no information about the solution of the model.

`Feasible` specifies that the algorithm found a feasible solution (that is, an assignment of values to variables that satisfies the constraints of the model, though it may not necessarily be optimal). The member functions `IloAlgorithm::getValue` access this feasible solution.

`Optimal` specifies that the algorithm found an optimal solution (that is, an assignment of values to variables that satisfies all the constraints of the model and that is proved optimal with respect to the objective of the model). The member functions `IloAlgorithm::getValue` access this optimal solution.

`Infeasible` specifies that the algorithm proved the model infeasible; that is, it is not possible to find an assignment of values to variables satisfying all the constraints in the model.

`Unbounded` specifies that the algorithm proved the model unbounded.

`InfeasibleOrUnbounded` specifies that the model is infeasible or unbounded.

`Error` specifies that an error occurred and, on platforms that support exceptions, that an exception has been thrown.

See Also: IloAlgorithm, operator<<

Fields:

`Unknown`

`Feasible`

`Optimal`

`Infeasible`

`Unbounded`

`InfeasibleOrUnbounded`

`Error`

Enumeration Type

Definition file: ilcplex/ilocplexi.h

This enumeration lists the types of bounds that may appear in a conflict.

Fields:

```
Lower
Upper
```

Enumeration Algorithm

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::Algorithm` lists the algorithms available in CPLEX to solve continuous models as controlled by the parameters `IloCplex::RootAlg` and `IloCplex::NodeAlg`.

These values are also returned by `IloCplex::getAlgorithm` to specify the algorithm used to generate the current solution. The values `FeasOpt` and `MIP` are returned by `IloCplex::getAlgorithm` but should **not** be used with `IloCplex::RootAlg` nor with `IloCplex::NodeAlg`.

See Also: IloCplex, IloCplex::getAlgorithm, IloCplex::getSubAlgorithm, IloCplex::IntParam, RootAlg, NodeAlg

Fields:

`NoAlg = CPX_ALG_NONE`	= CPX_ALG_NONE
`AutoAlg = CPX_ALG_AUTOMATIC`	= CPX_ALG_AUTOMATIC
`Primal = CPX_ALG_PRIMAL`	= CPX_ALG_PRIMAL
`Dual = CPX_ALG_DUAL`	= CPX_ALG_DUAL
`Barrier = CPX_ALG_BARRIER`	= CPX_ALG_BARRIER
`Sifting = CPX_ALG_SIFTING`	= CPX_ALG_SIFTING
`Concurrent = CPX_ALG_CONCURRENT`	= CPX_ALG_CONCURRENT
`Network = CPX_ALG_NET`	= CPX_ALG_NET
`FeasOpt = CPX_ALG_FEASOPT`	= CPX_ALG_FEASOPT
`MIP = CPX_ALG_MIP`	= CPX_ALG_MIP

Enumeration BasisStatus

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::BasisStatus` lists values that the status of variables or range constraints may assume in a basis. `NotABasicStatus` is not a valid status for a variable. A basis containing such a status does not constitute a valid basis. The basis status of a ranged constraint corresponds to the basis status of the corresponding slack or artificial variable that `IloCplex` manages for it. `FreeOrSuperbasic` specifies that the variable is nonbasic, but not at a bound.

See Also: IloCplex, IloCplex::BasisStatusArray

Fields:

```
NotABasicStatus = -1

Basic = 1

AtLower = 0

AtUpper = 2

FreeOrSuperbasic = 3
```

Enumeration BoolParam

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::BoolParam` lists the parameters of CPLEX that require Boolean values. Boolean values are also known in certain contexts as binary values or as zero-one (0-1) values. Use these values with the methods that accept Boolean parameters: `IloCplex::getParam` and `IloCplex::setParam`.

See the *CPLEX Parameters Reference Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`PreInd = CPX_PARAM_PREIND`	= CPX_PARAM_PREIND
`ReverseInd = CPX_PARAM_REVERSEIND`	= deprecated
`XXXInd = CPX_PARAM_XXXIND`	= deprecated
`MIPOrdInd = CPX_PARAM_MIPORDIND`	= CPX_PARAM_MIPORDIND
`MPSLongNum = CPX_PARAM_MPSLONGNUM`	= CPX_PARAM_MPSLONGNUM
`LBHeur = CPX_PARAM_LBHEUR`	= CPX_PARAM_LBHEUR
`PerInd = CPX_PARAM_PERIND`	= CPX_PARAM_PERIND
`PreLinear = CPX_PARAM_PRELINEAR`	= CPX_PARAM_PRELINEAR
`DataCheck = CPX_PARAM_DATACHECK`	= CPX_PARAM_DATACHECK
`QPmakePSDInd = CPX_PARAM_QPMAKEPSDIND`	= CPX_PARAM_QPMAKEPSDIND
`MemoryEmphasis = CPX_PARAM_MEMORYEMPHASIS`	= CPX_PARAM_MEMORYEMPHASIS
`NumericalEmphasis = CPX_PARAM_NUMERICALEMPHASIS`	= CPX_PARAM_NUMERICALEMPHASIS

Enumeration BranchDirection

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::BranchDirection` lists values that can be used for specifying branch directions either with the branch direction parameter `IloCplex::BrDir` or with the methods `IloCplex::setDirection` and `IloCplex::setDirections`. The branch direction specifies which direction to explore first after branching on one variable.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex, IloCplex::BranchDirectionArray

Fields:

`BranchGlobal = CPX_BRANCH_GLOBAL`	= CPX_BRANCH_GLOBAL
`BranchDown = CPX_BRANCH_DOWN`	= CPX_BRANCH_DOWN
`BranchUp = CPX_BRANCH_UP`	= CPX_BRANCH_UP

Enumeration ConflictStatus

Definition file: ilcplex/ilocplexi.h

This enumeration lists the values that tell the status of a constraint or bound with respect to a conflict.

- `ConflictPossibleMember`
- `ConflictMember`

The value `ConflictExcluded` is internal, undocumented, not available to users.

Fields:

```
ConflictExcluded = CPX_CONFLICT_EXCLUDED
```

```
ConflictPossibleMember = CPX_CONFLICT_POSSIBLE_MEMBER
```

```
ConflictMember = CPX_CONFLICT_MEMBER
```

Enumeration CplexStatus

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::CplexStatus` lists values that the status of an `IloCplex` algorithm can assume. The methods `IloCplex::getCplexStatus` and `IloCplex::getCplexSubStatus` access the status values, providing information about what the algorithm learned about the active model in the most recent invocation of the method `solve` or `feasOpt`. The status may also tell why the algorithm terminated.

See the group optim.cplex.solutionstatus in the *Callable Library Reference Manual*, where they are listed in alphabetic order, or the topic *Interpreting Solution Status Codes* in the Overview of the APIs, where they are listed in numeric order, for more information about these values. Also see the *CPLEX User's Manual* for examples of their use.

See also the enumeration `IloAlgorithm::Status` in the *Reference Manual* of the C++ API.

See Also: IloCplex

Fields:

`Unknown = 0`

`Optimal = CPX_STAT_OPTIMAL` = CPX_STAT_OPTIMAL

`Unbounded = CPX_STAT_UNBOUNDED` = CPX_STAT_UNBOUNDED

`Infeasible = CPX_STAT_INFEASIBLE` = CPX_STAT_INFEASIBLE

`InfOrUnbd = CPX_STAT_INForUNBD` = CPX_STAT_INForUNBD

`OptimalInfeas = CPX_STAT_OPTIMAL_INFEAS` = CPX_STAT_OPTIMAL_INFEAS

`NumBest = CPX_STAT_NUM_BEST` = CPX_STAT_NUM_BEST

`FeasibleRelaxedSum = CPX_STAT_FEASIBLE_RELAXED_SUM` = CPX_STAT_FEASIBLE_RELAXED_SUM

`OptimalRelaxedSum = CPX_STAT_OPTIMAL_RELAXED_SUM` = CPX_STAT_OPTIMAL_RELAXED_SUM

`FeasibleRelaxedInf = CPX_STAT_FEASIBLE_RELAXED_INF` = CPX_STAT_FEASIBLE_RELAXED_INF

`OptimalRelaxedInf = CPX_STAT_OPTIMAL_RELAXED_INF` = CPX_STAT_OPTIMAL_RELAXED_INF

`FeasibleRelaxedQuad = CPX_STAT_FEASIBLE_RELAXED_QUAD` = CPX_STAT_FEASIBLE_RELAXED_QUAD

`OptimalRelaxedQuad = CPX_STAT_OPTIMAL_RELAXED_QUAD` = CPX_STAT_OPTIMAL_RELAXED_QUAD

`AbortRelaxed = CPXMIP_ABORT_RELAXED` = CPXMIP_ABORT_RELAXED

`AbortObjLim = CPX_STAT_ABORT_OBJ_LIM` = CPX_STAT_ABORT_OBJ_LIM

`AbortPrimObjLim = CPX_STAT_ABORT_PRIM_OBJ_LIM` = CPX_STAT_ABORT_PRIM_OBJ_LIM

`AbortDualObjLim = CPX_STAT_ABORT_DUAL_OBJ_LIM` = CPX_STAT_ABORT_DUAL_OBJ_LIM

`AbortItLim = CPX_STAT_ABORT_IT_LIM` = CPX_STAT_ABORT_IT_LIM

`AbortTimeLim = CPX_STAT_ABORT_TIME_LIM` = CPX_STAT_ABORT_TIME_LIM

```
AbortUser = CPX_STAT_ABORT_USER                = CPX_STAT_ABORT_USER
OptimalFaceUnbounded =                         = CPX_STAT_OPTIMAL_FACE_UNBOUNDED
CPX_STAT_OPTIMAL_FACE_UNBOUNDED
OptimalTol = CPXMIP_OPTIMAL_TOL                = CPXMIP_OPTIMAL_TOL
SolLim = CPXMIP_SOL_LIM                        = CPXMIP_SOL_LIM
PopulateSolLim = CPXMIP_POPULATESOL_LIM        = CPXMIP_POPULATESOL_LIM
NodeLimFeas = CPXMIP_NODE_LIM_FEAS             = CPXMIP_NODE_LIM_FEAS
NodeLimInfeas = CPXMIP_NODE_LIM_INFEAS         = CPXMIP_NODE_LIM_INFEAS
FailFeas = CPXMIP_FAIL_FEAS                    = CPXMIP_FAIL_FEAS
FailInfeas = CPXMIP_FAIL_INFEAS                = CPXMIP_FAIL_INFEAS
MemLimFeas = CPXMIP_MEM_LIM_FEAS               = CPXMIP_MEM_LIM_FEAS
MemLimInfeas = CPXMIP_MEM_LIM_INFEAS           = CPXMIP_MEM_LIM_INFEAS
FailFeasNoTree =                               = CPXMIP_FAIL_FEAS_NO_TREE
CPXMIP_FAIL_FEAS_NO_TREE
FailInfeasNoTree =                             = CPXMIP_FAIL_INFEAS_NO_TREE
CPXMIP_FAIL_INFEAS_NO_TREE
ConflictFeasible =                             = CPX_STAT_CONFLICT_FEASIBLE
CPX_STAT_CONFLICT_FEASIBLE
ConflictMinimal =                              = CPX_STAT_CONFLICT_MINIMAL
CPX_STAT_CONFLICT_MINIMAL
ConflictAbortContradiction =                   =
CPX_STAT_CONFLICT_ABORT_CONTRADICTION          CPX_STAT_CONFLICT_ABORT_CONTRADICTION
ConflictAbortTimeLim =                         = CPX_STAT_CONFLICT_ABORT_TIME_LIM
CPX_STAT_CONFLICT_ABORT_TIME_LIM
ConflictAbortItLim =                           = CPX_STAT_CONFLICT_ABORT_IT_LIM
CPX_STAT_CONFLICT_ABORT_IT_LIM
ConflictAbortNodeLim =                         = CPX_STAT_CONFLICT_ABORT_NODE_LIM
CPX_STAT_CONFLICT_ABORT_NODE_LIM
ConflictAbortObjLim =                          = CPX_STAT_CONFLICT_ABORT_OBJ_LIM
CPX_STAT_CONFLICT_ABORT_OBJ_LIM
ConflictAbortMemLim =                          = CPX_STAT_CONFLICT_ABORT_MEM_LIM
CPX_STAT_CONFLICT_ABORT_MEM_LIM
ConflictAbortUser =                            = CPX_STAT_CONFLICT_ABORT_USER
CPX_STAT_CONFLICT_ABORT_USER
Feasible = CPX_STAT_FEASIBLE                   = CPX_STAT_FEASIBLE
OptimalPopulated =                             = CPXMIP_OPTIMAL_POPULATED
CPXMIP_OPTIMAL_POPULATED
OptimalPopulatedTol =                          = CPXMIP_OPTIMAL_POPULATED_TOL
CPXMIP_OPTIMAL_POPULATED_TOL
```

Enumeration CutType

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::CutType` lists the values that may be used in querying the number of cuts used in a mixed integer optimization with `getNcuts()`.

Fields:

```
CutCover = CPX_CUT_COVER

CutGubCover = CPX_CUT_GUBCOVER

CutFlowCover = CPX_CUT_FLOWCOVER

CutClique = CPX_CUT_CLIQUE

CutFrac = CPX_CUT_FRAC

CutMCF = CPX_CUT_MCF

CutMir = CPX_CUT_MIR

CutFlowPath = CPX_CUT_FLOWPATH

CutDisj = CPX_CUT_DISJ

CutImplBd = CPX_CUT_IMPLBD

CutZeroHalf = CPX_CUT_ZEROHALF

CutLocalCover = CPX_CUT_LOCALCOVER

CutTighten = CPX_CUT_TIGHTEN

CutObjDisj = CPX_CUT_OBJDISJ

CutUser = CPX_CUT_USER

CutTable = CPX_CUT_TABLE

CutSolnPool = CPX_CUT_SOLNPOOL
```

Enumeration DeleteMode

Definition file: ilcplex/ilocplexi.h

This enumeration lists the possible settings for the delete mode of `IloCplex` as controlled by the method `IloCplex::setDeleteMode` and queried by the method `IloCplex::getDeleteMode`.

- `IloCplex::LeaveBasis`

With the default setting `IloCplex::LeaveBasis`, an existing basis will remain unchanged if variables or constraints are removed from the loaded LP model. This choice generally renders the basis unusable for a restart when CPLEX is solving the modified LP and the advanced indicator (parameter `IloCplex::AdvInd`) is set to `IloTrue`.

- `IloCplex::FixBasis`

In contrast, with delete mode set to `IloCplex::FixBasis`, the invoking object will do basis pivots in order to maintain a valid basis when variables or constraints are removed. This choice makes the delete operation more computation-intensive, but may give a better starting point for reoptimization after modification of the extracted model.

If no basis is present in the invoking object, the setting of the delete mode has no effect.

See Also: IloCplex

Fields:

`LeaveBasis`

`FixBasis`

Enumeration DualPricing

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::DualPricing` lists values that the dual pricing parameter `IloCplex:DPriInd` can assume in `IloCplex` for use with the dual simplex algorithm. Use these values with the method `IloCplex::setParam(IloCplex::DPriInd, value)` when you set the dual pricing indicator.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`DPriIndAuto = CPX_DPRIIND_AUTO`	= CPX_DPRIIND_AUTO
`DPriIndFull = CPX_DPRIIND_FULL`	= CPX_DPRIIND_FULL
`DPriIndSteep = CPX_DPRIIND_STEEP`	= CPX_DPRIIND_STEEP
`DPriIndFullSteep = CPX_DPRIIND_FULLSTEEP`	= CPX_DPRIIND_FULLSTEEP
`DPriIndSteepQStart = CPX_DPRIIND_STEEPQSTART`	= CPX_DPRIIND_STEEPQSTART
`DPriIndDevex = CPX_DPRIIND_DEVEX`	= CPX_DPRIIND_DEVEX

Enumeration IntParam

Definition file: ilcplex/ilocplexi.h

`IloCplex` is the class for the algorithms in CPLEX. The enumeration `IloCplex::IntParam` lists the parameters of CPLEX that require integer values. Use these values with the methods `IloCplex::getParam` and `IloCplex::setParam`.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`AdvInd = CPX_PARAM_ADVIND`	= CPX_PARAM_ADVIND
`RootAlg = CPX_PARAM_LPMETHOD`	= CPX_PARAM_STARTALG, CPX_PARAM_LPMETHOD, CPX_PARAM_QPMETHOD
`NodeAlg = CPX_PARAM_SUBALG`	= CPX_PARAM_SUBALG
`MIPEmphasis = CPX_PARAM_MIPEMPHASIS`	= CPX_PARAM_MIPEMPHASIS
`AggFill = CPX_PARAM_AGGFILL`	= CPX_PARAM_AGGFILL
`AggInd = CPX_PARAM_AGGIND`	= CPX_PARAM_AGGIND
`BasInterval = CPX_PARAM_BASINTERVAL`	= CPX_PARAM_BASINTERVAL
`ClockType = CPX_PARAM_CLOCKTYPE`	= CPX_PARAM_CLOCKTYPE
`CraInd = CPX_PARAM_CRAIND`	= CPX_PARAM_CRAIND
`DepInd = CPX_PARAM_DEPIND`	= CPX_PARAM_DEPIND
`PreDual = CPX_PARAM_PREDUAL`	= CPX_PARAM_PREDUAL
`PrePass = CPX_PARAM_PREPASS`	= CPX_PARAM_PREPASS
`RelaxPreInd = CPX_PARAM_RELAXPREIND`	= CPX_PARAM_RELAXPREIND
`RepeatPresolve = CPX_PARAM_REPEATPRESOLVE`	= CPX_PARAM_REPEATPRESOLVE
`Symmetry = CPX_PARAM_SYMMETRY`	= CPX_PARAM_SYMMETRY
`DPriInd = CPX_PARAM_DPRIIND`	= CPX_PARAM_DPRIIND
`PriceLim = CPX_PARAM_PRICELIM`	= CPX_PARAM_PRICELIM
`SimDisplay = CPX_PARAM_SIMDISPLAY`	= CPX_PARAM_SIMDISPLAY
`ItLim = CPX_PARAM_ITLIM`	= CPX_PARAM_ITLIM
`NetFind = CPX_PARAM_NETFIND`	= CPX_PARAM_NETFIND
`PerLim = CPX_PARAM_PERLIM`	= CPX_PARAM_PERLIM
`PPriInd = CPX_PARAM_PPRIIND`	= CPX_PARAM_PPRIIND
`ReInv = CPX_PARAM_REINV`	= CPX_PARAM_REINV
`ScaInd = CPX_PARAM_SCAIND`	= CPX_PARAM_SCAIND
`Threads = CPX_PARAM_THREADS`	= CPX_PARAM_THREADS

```
ParallelMode = CPX_PARAM_PARALLELMODE        = CPX_PARAM_PARALLELMODE
SingLim = CPX_PARAM_SINGLIM                  = CPX_PARAM_SINGLIM
Reduce = CPX_PARAM_REDUCE                    = CPX_PARAM_REDUCE
NzReadLim = CPX_PARAM_NZREADLIM              = CPX_PARAM_NZREADLIM
ColReadLim = CPX_PARAM_COLREADLIM            = CPX_PARAM_COLREADLIM
RowReadLim = CPX_PARAM_ROWREADLIM            = CPX_PARAM_ROWREADLIM
QPNzReadLim = CPX_PARAM_QPNZREADLIM          = CPX_PARAM_QPNZREADLIM
WriteLevel = CPX_PARAM_WRITELEVEL            = CPX_PARAM_WRITELEVEL
SiftDisplay = CPX_PARAM_SIFTDISPLAY          = CPX_PARAM_SIFTDISPLAY
SiftAlg = CPX_PARAM_SIFTALG                  = CPX_PARAM_SIFTALG
SiftItLim = CPX_PARAM_SIFTITLIM              = CPX_PARAM_SIFTITLIM
BrDir = CPX_PARAM_BRDIR                      = CPX_PARAM_BRDIR
Cliques = CPX_PARAM_CLIQUES                  = CPX_PARAM_CLIQUES
CoeRedInd = CPX_PARAM_COEREDIND              = CPX_PARAM_COEREDIND
Covers = CPX_PARAM_COVERS                    = CPX_PARAM_COVERS
MIPDisplay = CPX_PARAM_MIPDISPLAY            = CPX_PARAM_MIPDISPLAY
MIPInterval = CPX_PARAM_MIPINTERVAL          = CPX_PARAM_MIPINTERVAL
IntSolLim = CPX_PARAM_INTSOLLIM              = CPX_PARAM_INTSOLLIM
NodeFileInd = CPX_PARAM_NODEFILEIND          = CPX_PARAM_NODEFILEIND
NodeLim = CPX_PARAM_NODELIM                  = CPX_PARAM_NODELIM
NodeSel = CPX_PARAM_NODESEL                  = CPX_PARAM_NODESEL
VarSel = CPX_PARAM_VARSEL                    = CPX_PARAM_VARSEL
BndStrenInd = CPX_PARAM_BNDSTRENIND          = CPX_PARAM_BNDSTRENIND
HeurFreq = CPX_PARAM_HEURFREQ                = CPX_PARAM_HEURFREQ
RINSHeur = CPX_PARAM_RINSHEUR                = CPX_PARAM_RINSHEUR
FPHeur = CPX_PARAM_FPHEUR                    = CPX_PARAM_FPHEUR
RepairTries = CPX_PARAM_REPAIRTRIES          = CPX_PARAM_REPAIRTRIES
SubMIPNodeLim = CPX_PARAM_SUBMIPNODELIM      = CPX_PARAM_SUBMIPNODELIM
MIPOrdType = CPX_PARAM_MIPORDTYPE            = CPX_PARAM_MIPORDTYPE
BBInterval = CPX_PARAM_BBINTERVAL            = CPX_PARAM_BBINTERVAL
FlowCovers = CPX_PARAM_FLOWCOVERS            = CPX_PARAM_FLOWCOVERS
ImplBd = CPX_PARAM_IMPLBD                    = CPX_PARAM_IMPLBD
Probe = CPX_PARAM_PROBE                      = CPX_PARAM_PROBE
GUBCovers = CPX_PARAM_GUBCOVERS              = CPX_PARAM_GUBCOVERS
StrongCandLim = CPX_PARAM_STRONGCANDLIM      = CPX_PARAM_STRONGCANDLIM
StrongItLim = CPX_PARAM_STRONGITLIM          = CPX_PARAM_STRONGITLIM
```

```
FracCand = CPX_PARAM_FRACCAND                = CPX_PARAM_FRACCAND
FracCuts = CPX_PARAM_FRACCUTS                = CPX_PARAM_FRACCUTS
FracPass = CPX_PARAM_FRACPASS                = CPX_PARAM_FRACPASS
PreslvNd = CPX_PARAM_PRESLVND                = CPX_PARAM_PRESLVND
FlowPaths = CPX_PARAM_FLOWPATHS              = CPX_PARAM_FLOWPATHS
MIRCuts = CPX_PARAM_MIRCUTS                  = CPX_PARAM_MIRCUTS
DisjCuts = CPX_PARAM_DISJCUTS                = CPX_PARAM_DISJCUTS
ZeroHalfCuts = CPX_PARAM_ZEROHALFCUTS        = CPX_PARAM_ZEROHALFCUTS
MCFCuts = CPX_PARAM_MCFCUTS                  = CPX_PARAM_MCFCUTS
AggCutLim = CPX_PARAM_AGGCUTLIM              = CPX_PARAM_AGGCUTLIM
CutPass = CPX_PARAM_CUTPASS                  = CPX_PARAM_CUTPASS
EachCutLim = CPX_PARAM_EACHCUTLIM            = CPX_PARAM_EACHCUTLIM
DiveType = CPX_PARAM_DIVETYPE                = CPX_PARAM_DIVETYPE
MIPSearch = CPX_PARAM_MIPSEARCH              = CPX_PARAM_MIPSEARCH
MIQCPStrat = CPX_PARAM_MIQCPSTRAT            = CPX_PARAM_MIQCPSTRAT
SolnPoolCapacity =                           = CPX_PARAM_SOLNPOOLCAPACITY
CPX_PARAM_SOLNPOOLCAPACITY
SolnPoolReplace = CPX_PARAM_SOLNPOOLREPLACE  = CPX_PARAM_SOLNPOOLREPLACE
SolnPoolIntensity =                          = CPX_PARAM_SOLNPOOLINTENSITY
CPX_PARAM_SOLNPOOLINTENSITY
PopulateLim = CPX_PARAM_POPULATELIM          = CPX_PARAM_POPULATELIM
BarAlg = CPX_PARAM_BARALG                    = CPX_PARAM_BARALG
BarColNz = CPX_PARAM_BARCOLNZ                = CPX_PARAM_BARCOLNZ
BarDisplay = CPX_PARAM_BARDISPLAY            = CPX_PARAM_BARDISPLAY
BarItLim = CPX_PARAM_BARITLIM                = CPX_PARAM_BARITLIM
BarMaxCor = CPX_PARAM_BARMAXCOR              = CPX_PARAM_BARMAXCOR
BarOrder = CPX_PARAM_BARORDER                = CPX_PARAM_BARORDER
BarCrossAlg = CPX_PARAM_BARCROSSALG          = CPX_PARAM_BARCROSSALG
BarStartAlg = CPX_PARAM_BARSTARTALG          = CPX_PARAM_BARSTARTALG
NetItLim = CPX_PARAM_NETITLIM                = CPX_PARAM_NETITLIM
NetPPriInd = CPX_PARAM_NETPPRIIND            = CPX_PARAM_NETPPRIIND
NetDisplay = CPX_PARAM_NETDISPLAY            = CPX_PARAM_NETDISPLAY
ConflictDisplay = CPX_PARAM_CONFLICTDISPLAY  = CPX_PARAM_CONFLICTDISPLAY
FeasOptMode = CPX_PARAM_FEASOPTMODE          = CPX_PARAM_FEASOPTMODE
TuningMeasure = CPX_PARAM_TUNINGMEASURE      = CPX_PARAM_TUNINGMEASURE
TuningRepeat = CPX_PARAM_TUNINGREPEAT        = CPX_PARAM_TUNINGREPEAT
```

```
TuningDisplay = CPX_PARAM_TUNINGDISPLAY       = CPX_PARAM_TUNINGDISPLAY
PolishAfterNode = CPX_PARAM_POLISHAFTERNODE   = CPX_PARAM_POLISHAFTERNODE
PolishAfterIntSol =                           = CPX_PARAM_POLISHAFTERINTSOL
CPX_PARAM_POLISHAFTERINTSOL
```

Enumeration MIPEmphasisType

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::MIPEmphasisType` lists the values that the MIP emphasis parameter `IloCplex::MIPEmphasis` can assume in an instance of `IloCplex` for use when it is solving MIP problems. Use these values with the method `IloCplex::setParam(IloCplex::MIPEmphasis, value)` when you set MIP emphasis.

With the default setting of `IloCplex::MIPEmphasisBalance`, `IloCplex` tries to compute the branch-and-cut algorithm in such a way as to find a proven optimal solution quickly. For a discussion about various settings, refer to the *CPLEX User's Manual*.

See the *CPLEX Parameters Reference Manual* and the information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`MIPEmphasisBalanced = CPX_MIPEMPHASIS_BALANCED`	= CPX_MIPEMPHASIS_BALANCED
`MIPEmphasisOptimality = CPX_MIPEMPHASIS_OPTIMALITY`	= CPX_MIPEMPHASIS_OPTIMALITY
`MIPEmphasisFeasibility = CPX_MIPEMPHASIS_FEASIBILITY`	= CPX_MIPEMPHASIS_FEASIBILITY
`MIPEmphasisBestBound = CPX_MIPEMPHASIS_BESTBOUND`	= CPX_MIPEMPHASIS_BESTBOUND
`MIPEmphasisHiddenFeas = CPX_MIPEMPHASIS_HIDDENFEAS`	= CPX_MIPEMPHASIS_HIDDENFEAS

Enumeration MIPStartEffort

Definition file: ilcplex/ilocplexi.h

This enumeration defines the possible levels of effort for CPLEX to expend when CPLEX evaluates a MIP start.

- Level 0 (zero) MIPStartAuto: Automatic, let CPLEX decide.
- Level 1 (one) MIPStartCheckFeas: CPLEX checks the feasibility of the MIP start.
- Level 2 MIPStartSolveFixed: CPLEX solves the fixed problem specified by the MIP start.
- Level 3 MIPStartSolveMIP: CPLEX solves a subMIP.
- Level 4 MIPStartRepair:CPLEX attempts to repair the MIP start if it is infeasible, according to the parameter that sets the frequency to try to repair infeasible MIP start, `CPX_PARAM_REPAIRTRIES`.

Fields:

```
MIPStartAuto = CPX_MIPSTART_AUTO

MIPStartCheckFeas = CPX_MIPSTART_CHECKFEAS

MIPStartSolveFixed = CPX_MIPSTART_SOLVEFIXED

MIPStartSolveMIP = CPX_MIPSTART_SOLVEMIP

MIPStartRepair = CPX_MIPSTART_REPAIR
```

Enumeration MIPsearch

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::MIPsearch` lists values that the dynamic search parameter `IloCplex::MIPSearch` can assume in `IloCplex`. Use these values with the method IloCplex::setParam `(IloCplex::MIPSearch, value)`.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about this parameter. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`AutoSearch = CPX_MIPSEARCH_AUTO`	= CPX_MIPSEARCH_AUTO
`Traditional = CPX_MIPSEARCH_TRADITIONAL`	= CPX_MIPSEARCH_TRADITIONAL
`Dynamic = CPX_MIPSEARCH_DYNAMIC`	= CPX_MIPSEARCH_DYNAMIC

Enumeration NodeSelect

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::NodeSelect` lists values that the parameter `IloCplex::NodeSel` can assume in `IloCplex`. Use these values with the method `IloCplex::setParam(IloCplex::NodeSel, value)`.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`DFS = CPX_NODESEL_DFS`	= CPX_NODESEL_DFS
`BestBound = CPX_NODESEL_BESTBOUND`	= CPX_NODESEL_BESTBOUND
`BestEst = CPX_NODESEL_BESTEST`	= CPX_NODESEL_BESTEST
`BestEstAlt = CPX_NODESEL_BESTEST_ALT`	= CPX_NODESEL_BESTEST_ALT

Enumeration NumParam

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::NumParam` lists the parameters of CPLEX that require numeric values. Use these values with the member functions: `IloCplex::getParam` and `IloCplex::setParam`.

See the *CPLEX Parameters Reference Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for more examples of their use.

See Also: IloCplex

Fields:

`EpMrk = CPX_PARAM_EPMRK`	= CPX_PARAM_EPMRK
`EpOpt = CPX_PARAM_EPOPT`	= CPX_PARAM_EPOPT
`EpPer = CPX_PARAM_EPPER`	= CPX_PARAM_EPPER
`EpRHS = CPX_PARAM_EPRHS`	= CPX_PARAM_EPRHS
`NetEpOpt = CPX_PARAM_NETEPOPT`	= CPX_PARAM_NETEPOPT
`NetEpRHS = CPX_PARAM_NETEPRHS`	= CPX_PARAM_NETEPRHS
`TiLim = CPX_PARAM_TILIM`	= CPX_PARAM_TILIM
`TuningTiLim = CPX_PARAM_TUNINGTILIM`	= CPX_PARAM_TUNINGTILIM
`BtTol = CPX_PARAM_BTTOL`	= CPX_PARAM_BTTOL
`CutLo = CPX_PARAM_CUTLO`	= CPX_PARAM_CUTLO
`CutUp = CPX_PARAM_CUTUP`	= CPX_PARAM_CUTUP
`EpGap = CPX_PARAM_EPGAP`	= CPX_PARAM_EPGAP
`EpInt = CPX_PARAM_EPINT`	= CPX_PARAM_EPINT
`EpAGap = CPX_PARAM_EPAGAP`	= CPX_PARAM_EPAGAP
`EpRelax = CPX_PARAM_EPRELAX`	= CPX_PARAM_EPRELAX
`ObjDif = CPX_PARAM_OBJDIF`	= CPX_PARAM_OBJDIF
`ObjLLim = CPX_PARAM_OBJLLIM`	= CPX_PARAM_OBJLLIM
`ObjULim = CPX_PARAM_OBJULIM`	= CPX_PARAM_OBJULIM
`PolishTime = CPX_PARAM_POLISHTIME`	= CPX_PARAM_POLISHTIME
`PolishAfterEpAGap = CPX_PARAM_POLISHAFTEREPAGAP`	= CPX_PARAM_POLISHAFTEREPAGAP
`PolishAfterEpGap = CPX_PARAM_POLISHAFTEREPGAP`	= CPX_PARAM_POLISHAFTEREPGAP
`PolishAfterTime = CPX_PARAM_POLISHAFTERTIME`	= CPX_PARAM_POLISHAFTERTIME
`ProbeTime = CPX_PARAM_PROBETIME`	= CPX_PARAM_PROBETIME
`RelObjDif = CPX_PARAM_RELOBJDIF`	= CPX_PARAM_RELOBJDIF
`CutsFactor = CPX_PARAM_CUTSFACTOR`	= CPX_PARAM_CUTSFACTOR
`TreLim = CPX_PARAM_TRELIM`	= CPX_PARAM_TRELIM
`SolnPoolGap = CPX_PARAM_SOLNPOOLGAP`	= CPX_PARAM_SOLNPOOLGAP

```
SolnPoolAGap = CPX_PARAM_SOLNPOOLAGAP          = CPX_PARAM_SOLNPOOLAGAP
WorkMem = CPX_PARAM_WORKMEM                    = CPX_PARAM_WORKMEM
BarEpComp = CPX_PARAM_BAREPCOMP                = CPX_PARAM_BAREPCOMP
BarQCPEpComp = CPX_PARAM_BARQCPEPCOMP          = CPX_PARAM_BARQCPEPCOMP
BarGrowth = CPX_PARAM_BARGROWTH                = CPX_PARAM_BARGROWTH
BarObjRng = CPX_PARAM_BAROBJRNG                = CPX_PARAM_BAROBJRNG
EpLin = CPX_PARAM_EPLIN                        = CPX_PARAM_EPLIN
```

Enumeration Parallel_Mode

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::ParallelMode` lists values that the parallel mode parameter `IloCplex::ParallelMode` can assume in `IloCplex` for use on multiprocessor or multithread platforms, if your application is licensed for parallel optimization. Use these values with the method IloCplex::setParam `(IloCplex::ParallelMode, value)`.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about this parameter. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`Opportunistic = CPX_PARALLEL_OPPORTUNISTIC`	= CPX_PARALLEL_OPPORTUNISTIC
`AutoParallel = CPX_PARALLEL_AUTO`	= CPX_PARALLEL_AUTO
`Deterministic = CPX_PARALLEL_DETERMINISTIC`	= CPX_PARALLEL_DETERMINISTIC

Enumeration PrimalPricing

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::PrimalPricing` lists values that the primal pricing parameter `IloCplex::PPriInd` can assume in `IloCplex` for use with the primal simplex algorithm. Use these values with the method IloCplex::setParam `(IloCplex::PPriInd, value)` when setting the primal pricing indicator.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`PPriIndPartial = CPX_PPRIIND_PARTIAL`	= CPX_PPRIIND_PARTIAL
`PPriIndAuto = CPX_PPRIIND_AUTO`	= CPX_PPRIIND_AUTO
`PPriIndDevex = CPX_PPRIIND_DEVEX`	= CPX_PPRIIND_DEVEX
`PPriIndSteep = CPX_PPRIIND_STEEP`	= CPX_PPRIIND_STEEP
`PPriIndSteepQStart = CPX_PPRIIND_STEEPQSTART`	= CPX_PPRIIND_STEEPQSTART
`PPriIndFull = CPX_PPRIIND_FULL`	= CPX_PPRIIND_FULL

Enumeration Quality

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::Quality` lists types of quality measures that can be queried for a solution with the method `IloCplex::getQuality`.

See the group optim.cplex.solutionquality in the *Callable Library Reference Manual* for more information about these values. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`MaxPrimalInfeas = CPX_MAX_PRIMAL_INFEAS`	= CPX_MAX_PRIMAL_INFEAS
`MaxScaledPrimalInfeas = CPX_MAX_SCALED_PRIMAL_INFEAS`	= CPX_MAX_SCALED_PRIMAL_INFEAS
`SumPrimalInfeas = CPX_SUM_PRIMAL_INFEAS`	= CPX_SUM_PRIMAL_INFEAS
`SumScaledPrimalInfeas = CPX_SUM_SCALED_PRIMAL_INFEAS`	= CPX_SUM_SCALED_PRIMAL_INFEAS
`MaxDualInfeas = CPX_MAX_DUAL_INFEAS`	= CPX_MAX_DUAL_INFEAS
`MaxScaledDualInfeas = CPX_MAX_SCALED_DUAL_INFEAS`	= CPX_MAX_SCALED_DUAL_INFEAS
`SumDualInfeas = CPX_SUM_DUAL_INFEAS`	= CPX_SUM_DUAL_INFEAS
`SumScaledDualInfeas = CPX_SUM_SCALED_DUAL_INFEAS`	= CPX_SUM_SCALED_DUAL_INFEAS
`MaxIntInfeas = CPX_MAX_INT_INFEAS`	= CPX_MAX_INT_INFEAS
`SumIntInfeas = CPX_SUM_INT_INFEAS`	= CPX_SUM_INT_INFEAS
`MaxPrimalResidual = CPX_MAX_PRIMAL_RESIDUAL`	= CPX_MAX_PRIMAL_RESIDUAL
`MaxScaledPrimalResidual = CPX_MAX_SCALED_PRIMAL_RESIDUAL`	= CPX_MAX_SCALED_PRIMAL_RESIDUAL
`SumPrimalResidual = CPX_SUM_PRIMAL_RESIDUAL`	= CPX_SUM_PRIMAL_RESIDUAL
`SumScaledPrimalResidual = CPX_SUM_SCALED_PRIMAL_RESIDUAL`	= CPX_SUM_SCALED_PRIMAL_RESIDUAL
`MaxDualResidual = CPX_MAX_DUAL_RESIDUAL`	= CPX_MAX_DUAL_RESIDUAL
`MaxScaledDualResidual = CPX_MAX_SCALED_DUAL_RESIDUAL`	= CPX_MAX_SCALED_DUAL_RESIDUAL
`SumDualResidual = CPX_SUM_DUAL_RESIDUAL`	= CPX_SUM_DUAL_RESIDUAL
`SumScaledDualResidual = CPX_SUM_SCALED_DUAL_RESIDUAL`	= CPX_SUM_SCALED_DUAL_RESIDUAL
`MaxCompSlack = CPX_MAX_COMP_SLACK`	= CPX_MAX_COMP_SLACK
`SumCompSlack = CPX_SUM_COMP_SLACK`	= CPX_SUM_COMP_SLACK
`MaxX = CPX_MAX_X`	= CPX_MAX_X
`MaxScaledX = CPX_MAX_SCALED_X`	= CPX_MAX_SCALED_X

```
MaxPi = CPX_MAX_PI                          = CPX_MAX_PI
MaxScaledPi = CPX_MAX_SCALED_PI             = CPX_MAX_SCALED_PI
MaxSlack = CPX_MAX_SLACK                    = CPX_MAX_SLACK
MaxScaledSlack = CPX_MAX_SCALED_SLACK       = CPX_MAX_SCALED_SLACK
MaxRedCost = CPX_MAX_RED_COST               = CPX_MAX_RED_COST
MaxScaledRedCost = CPX_MAX_SCALED_RED_COST  = CPX_MAX_SCALED_RED_COST
SumX = CPX_SUM_X                            = CPX_SUM_X
SumScaledX = CPX_SUM_SCALED_X               = CPX_SUM_SCALED_X
SumPi = CPX_SUM_PI                          = CPX_SUM_PI
SumScaledPi = CPX_SUM_SCALED_PI             = CPX_SUM_SCALED_PI
SumSlack = CPX_SUM_SLACK                    = CPX_SUM_SLACK
SumScaledSlack = CPX_SUM_SCALED_SLACK       = CPX_SUM_SCALED_SLACK
SumRedCost = CPX_SUM_RED_COST               = CPX_SUM_RED_COST
SumScaledRedCost = CPX_SUM_SCALED_RED_COST  = CPX_SUM_SCALED_RED_COST
Kappa = CPX_KAPPA                           = CPX_KAPPA
ObjGap = CPX_OBJ_GAP                        = CPX_OBJ_GAP
DualObj = CPX_DUAL_OBJ                      = CPX_DUAL_OBJ
PrimalObj = CPX_PRIMAL_OBJ                  = CPX_PRIMAL_OBJ
ExactKappa = CPX_EXACT_KAPPA                = CPX_EXACT_KAPPA
```

Enumeration Relaxation

Definition file: ilcplex/ilocplexi.h

The enumeration `Relaxation` lists the values that can be taken by the parameter `FeasOptMode`. This parameter controls several aspects of how the method `feasOpt` performs its relaxation.

The method `feasOpt` works in two phases. In its first phase, it attempts to find a minimum-penalty relaxation of a given infeasible model. If you want `feasOpt` to stop after this first phase, choose a value with `Min` in its symbolic name. If you want `feasOpt` to continue beyond its first phase to find a solution that is optimal with respect to the original objective function, subject to the constraint that the penalty of the relaxation must not exceed the value found in the first phase, then choose a value with `Opt` in its symbolic name.

In both phases, the suffixes `Sum`, `Inf`, and `Quad` specify the relaxation metric:

- `Sum` tells `feasOpt` to mimimize the weighted sum of the required relaxations of bounds and constraints according to the formula penalty = sum (penalty_i times relaxation_amount_i)
- `Inf` tells `feasOpt` to minimize the weighted number of bounds and constraints that are relaxed according to the formula penalty = sum (penalty_i times relaxation_indicator_i)
- `Quad` tells `feasOpt` to mimimize the weighted sum of the squares of required relaxations of bounds and constraints according to the formula penalty = sum (penalty_i times relaxation_amount_i times relaxation_amount_i)

Weights are determined by the preference values you provide as input to the method `feasOpt`.

When `IloAnd` is used to group constraints as input to `feasOpt`, the relaxation penalty is computed on groups instead of on individual constraints. For example, all constraints in a group can be relaxed for a total penalty of one unit under the various `Inf` metrics.

Fields:

`MinSum = CPX_FEASOPT_MIN_SUM`	= CPX_FEASOPT_MIN_SUM
`OptSum = CPX_FEASOPT_OPT_SUM`	= CPX_FEASOPT_OPT_SUM
`MinInf = CPX_FEASOPT_MIN_INF`	= CPX_FEASOPT_MIN_INF
`OptInf = CPX_FEASOPT_OPT_INF`	= CPX_FEASOPT_OPT_INF
`MinQuad = CPX_FEASOPT_MIN_QUAD`	= CPX_FEASOPT_MIN_QUAD
`OptQuad = CPX_FEASOPT_OPT_QUAD`	= CPX_FEASOPT_OPT_QUAD

Enumeration StringParam

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::StringParam` lists the parameters of CPLEX that require a character string as a value. Use these values with the methods `IloCplex::getParam` and `IloCplex::setParam`.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`WorkDir = CPX_PARAM_WORKDIR`	= CPX_PARAM_WORKDIR

Enumeration TuningStatus

Definition file: ilcplex/ilocplexi.h

This enumeration lists the values that are returned by `tuneParam`.

- `TuningComplete`
- `TuningAbort`
- `TuningTimeLim`

The value `ConflictExcluded` is internal, undocumented, not available to users.

Fields:

```
TuningComplete = 0
```

```
TuningAbort = CPX_TUNE_ABORT
```

```
TuningTimeLim = CPX_TUNE_TILIM
```

Enumeration VariableSelect

Definition file: ilcplex/ilocplexi.h

The enumeration `IloCplex::VariableSelect` lists values that the parameter `IloCplex::VarSel` can assume in `IloCplex`. Use these values with the method `IloCplex::setParam(IloCplex::VarSel, value)`.

See the *CPLEX Parameters Reference Manual* and the *CPLEX User's Manual* for more information about these parameters. Also see the *CPLEX User's Manual* for examples of their use.

See Also: IloCplex

Fields:

`MinInfeas = CPX_VARSEL_MININFEAS`	= CPX_VARSEL_MININFEAS
`DefaultVarSel = CPX_VARSEL_DEFAULT`	= CPX_VARSEL_DEFAULT
`MaxInfeas = CPX_VARSEL_MAXINFEAS`	= CPX_VARSEL_MAXINFEAS
`Pseudo = CPX_VARSEL_PSEUDO`	= CPX_VARSEL_PSEUDO
`Strong = CPX_VARSEL_STRONG`	= CPX_VARSEL_STRONG
`PseudoReduced = CPX_VARSEL_PSEUDOREDUCED`	= CPX_VARSEL_PSEUDOREDUCED

Enumeration WriteLevelType

Definition file: ilcplex/ilocplexi.h

Values of this enumeration specify how much detail CPLEX includes when it writes MIP starts and solutions to a formatted file.

Fields:

```
Auto = CPX_WRITELEVEL_AUTO
```

```
AllVars = CPX_WRITELEVEL_ALLVARS
```

```
DiscreteVars = CPX_WRITELEVEL_DISCRETEVARS
```

```
NonzeroVars = CPX_WRITELEVEL_NONZEROVARS
```

```
NonZeroDiscreteVars = CPX_WRITELEVEL_NONZERODISCRETEVARS
```

Enumeration Type

Definition file: ilconcert/iloexpression.h

An enumeration for the class IloNumVar.
This nested enumeration enables you to specify whether an instance of `IloNumVar` is of type integer (`Int`), Boolean (`Bool`), or floating-point (`Float`).

Programming Hint

For each enumerated value in `IloNumVar::Type`, there is a predefined equivalent C++ `#define` in the Concert Technology include files to make programming easier. For example, instead of writing `IloNumVar::Int` in your application, you can write `ILOINT`. Likewise, `ILOFLOAT` is defined for `IloNumVar::Float` and `ILOBOOL` for `IloNumVar::Bool`.

See Also: IloNumVar

Fields:

```
Int = 1

Float = 2

Bool = 3
```

Enumeration Sense

Definition file: ilconcert/ilolinear.h

Specifies objective as minimization or maximization.
An instance of the class `IloObjective` represents an objective in a model. This nested enumeration is limited in scope to that class, and its values specify the sense of an objective; that is, whether it is a minimization (`Minimize`) or a maximization (`Maximize`).

See Also: IloObjective

Fields:

```
Minimize = 1
```

```
Maximize = -1
```

Global function IloPiecewiseLinear

```
public IloNumExprArg IloPiecewiseLinear(const IloNumExprArg node, const IloNumArray
point, const IloNumArray slope, IloNum a, IloNum fa)
public IloNumExprArg IloPiecewiseLinear(const IloNumExprArg node, IloNum
firstSlope, const IloNumArray point, const IloNumArray value, IloNum lastSlope)
```

Definition file: ilconcert/iloexpression.h

Represents a continuous or discontinuous piecewise linear function.
The function `IloPiecewiseLinear` creates an expression node to represent a continuous or discontinuous piecewise linear function *f* of the variable *x*. The signatures of this overloaded function support two different ways to specify piecewise linear functions. One approach specifies the breakpoints and slopes of the segments of the PWL function. The other approach specifies the breakpoints and function values of the segments. Both approaches can specify either continuous or discontinuous piecewise linear functions.

However, the user must take care with the approach that uses breakpoints and slopes to specify a discontinuous piecewise linear function **uniquely**. For further explanation about unique specification of discontinuous piecewise linear functions, see the topic *Discontinuous piecewise linear functions* in the *IBM ILOG CPLEX User's Manual*.

In the approach using breakpoints and slopes to specify a PWL function, the array `point` contains the *n* breakpoints of the function such that *point[i-1] <= point[i] for i = 1, . . . , n-1*. The array `slope` contains the *n+1* slopes of the *n+1* segments of the function. The values `a` and `fa` must be coordinates of a point such that `fa = f(a)`.

The argument `a` cannot be the coordinate of a step. In other words, `a` must **not** be the `point[i]` if `point[i] == point[i-1]` or if `point[i] == point[i+1]`.

Example

```
IloPiecewiseLinear(x, IloNumArray(env, 2, 10., 20.),
                   IloNumArray(env, 3, 0.3, 1., 2.),
                   0, 0);
```

That expression defines a piecewise linear function *f* having two breakpoints at *x = 10* and *x = 20*, and three segments; their slopes are *0.3, 1,* and *2*. The first segment has infinite length and ends at the point *(x = 10, f(x) = 3)* since *f(0) = 0*. The second segment starts at the point *(x = 10, f(x) = 3)* and ends at the point *(x = 20, f(x) = 13)*, where the third segment starts.

For the approach that defines a piecewise linear function by breakpoints and values, the array `point` also contains the *n* breakpoints of the PWL function. The array `value` contains the corresponding *n* values of the function. The argument `firstSlope` specifies the slope of the segment to the left of the first breakpoint; the argument `lastSlope` specifies the sleop of the segment to the right of the final breakpoint.

For an example and further explanation of this approach of specification by breakpoint and value, see the topic *Discontinuous piecewise linear functions* in the *IBM ILOG CPLEX User's Manual*.

Global function operator>=

```
public IloConstraint operator>=(IloNumExprArg base, IloNumExprArg base2)
public IloRange operator>=(IloNumExprArg expr, IloNum val)
public IloRange operator>=(IloNum val, IloNumExprArg eb)
```

Definition file: ilconcert/ilolinear.h

overloaded C++ operator
This overloaded C++ operator constrains its first argument to be greater than or equal to its second argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function IloArcCos

```
public IloNumExprArg IloArcCos(const IloNumExprArg arg)
public IloNum IloCos(IloNum val)
public IloNum IloSin(IloNum val)
public IloNum IloTan(IloNum val)
public IloNum IloArcCos(IloNum val)
public IloNum IloArcSin(IloNum val)
public IloNum IloArcTan(IloNum val)
public IloNumExprArg IloSin(const IloNumExprArg arg)
public IloNumExprArg IloCos(const IloNumExprArg arg)
public IloNumExprArg IloTan(const IloNumExprArg arg)
public IloNumExprArg IloArcSin(const IloNumExprArg arg)
public IloNumExprArg IloArcTan(const IloNumExprArg arg)
```

Definition file: ilconcert/iloexpression.h

Trigonometric functions.
Concert Technology offers predefined functions that return an expression from a trigonometric function on an expression. These predefined functions also return a numeric value from a trigonometric function on a numeric value as well.

Programming Hint

If you want to manipulate constrained floating-point expressions in degrees, we strongly recommend that you call the trigonometric functions on variables expressed in radians and then convert the results to degrees (rather than declaring the constrained floating-point expressions in degrees and then converting them to radians to call the trigonometric functions).

The reason for that advice is that the method we recommend gives more accurate results in the context of the usual floating-point pitfalls.

Global function IloEndMT

```
public void IloEndMT()
```

Definition file: ilconcert/iloenv.h

Ends multithreading.
This function ends multithreading in a Concert Technology application.

Global function operator!=

```
public IloDiff operator!=(IloNumExprArg arg1, IloNumExprArg arg2)
public IloDiff operator!=(IloNumExprArg arg, IloNum val)
public IloDiff operator!=(IloNum val, IloNumExprArg arg)
```

Definition file: ilconcert/ilomodel.h

Overloaded C++ operator.
This overloaded C++ operator constrains its two arguments to be unequal (that is, different from each other). In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function operator>>

```
public istream & operator>>(istream & in, IloNumArray & a)
public istream & operator>>(istream & in, IloIntArray & a)
```

Definition file: ilconcert/iloenv.h

Overloaded C++ operator redirects input.
This overloaded C++ operator directs input to an input stream.

Global function IloFloor

```
public IloNum IloFloor(IloNum val)
```

Definition file: ilconcert/iloenv.h

Returns the largest integer value not greater than the argument.
This function computes the largest integer value not greater than `val`.

Examples:

```
IloFloor(IloInfinity) is IloInfinity.
IloFloor(-IloInfinity) is -IloInfinity.
IloFloor(0) is 0.
IloFloor(0.4) is 0.
IloFloor(-0.4) is -1.
IloFloor(0.5) is 0.
IloFloor(-0.5) is -1.
IloFloor(0.6) is 0.
IloFloor(-0.6) is -1.
```

Global function operator/

```
public IloNumExprArg operator/(const IloNumExprArg x, const IloNumExprArg y)
public IloNumExprArg operator/(const IloNumExprArg x, IloNum y)
public IloNumExprArg operator/(IloNum x, const IloNumExprArg y)
```

Definition file: ilconcert/iloexpression.h

Returns an expression equal to the quotient of its arguments.
This overloaded C++ operator returns an expression equal to the quotient of its arguments. Its arguments may be numeric values or numeric variables. For integer division, use IloDiv.

Global function operator>

```
public IloConstraint operator>(IloNumExprArg base, IloNumExprArg base2)
public IloConstraint operator>(IloNumExprArg base, IloNum val)
public IloConstraint operator>(IloNum val, IloNumExprArg eb)
public IloConstraint operator>(IloIntExprArg base, IloIntExprArg base2)
public IloConstraint operator>(IloIntExprArg base, IloInt val)
public IloConstraint operator>(IloInt val, IloIntExprArg eb)
```

Definition file: ilconcert/ilolinear.h

overloaded C++ operator
This overloaded C++ operator constrains its first argument to be strictly greater than its second argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function IloAbs

```
public IloNumExprArg IloAbs(const IloNumExprArg arg)
public IloNum IloAbs(IloNum val)
public IloNum IloPower(IloNum val1, IloNum val2)
public IloIntExprArg IloAbs(const IloIntExprArg arg)
```

Definition file: ilconcert/iloexpression.h

Returns the absolute value of its argument.
Concert Technology offers predefined functions that return an expression from an algebraic function on expressions. These predefined functions also return a numeric value from an algebraic function on numeric values as well.

`IloAbs` returns the absolute value of its argument.

What Is Extracted

`IloAbs` is extracted by an instance of `IloCplex` and linearized automatically.

`IloAbs` is extracted by an instance of `IloCP` or `IloSolver` as an instance of `IlcAbs`.

Global function IloCeil

```
public IloNum IloCeil(IloNum val)
```

Definition file: ilconcert/iloenv.h

Returns the least integer value not less than its argument.
This function computes the least integer value not less than `val`.

Examples:

```
IloCeil(IloInfinity) is IloInfinity.
IloCeil(-IloInfinity) is -IloInfinity.
IloCeil(0) is 0.
IloCeil(0.4) is 1.
IloCeil(-0.4) is 0.
IloCeil(0.5) is 1.
IloCeil(-0.5) is 0.
IloCeil(0.6) is 1.
IloCeil(-0.6) is 0.
```

Global function IloPower

```
public IloNumExprArg IloPower(const IloNumExprArg base, const IloNumExprArg
exponent)
public IloNumExprArg IloPower(const IloNumExprArg base, IloNum exponent)
public IloNumExprArg IloPower(IloNum base, const IloNumExprArg exponent)
```

Definition file: ilconcert/iloexpression.h

Returns the power of its arguments.
Concert Technology offers predefined functions that return an expression from an algebraic function over expressions. These predefined functions also return a numeric value from an algebraic function over numeric values as well.

`IloPower` returns the result of raising its `base` argument to the power of its `exponent` argument, that is, `base`**`exponent`. If `base` is a floating-point value or variable, then `exponent` must be greater than or equal to 0 (zero).

What Is Extracted

An instance of `IloCplex` can extract only quadratic terms that are positive semi-definite when they appear in an objective function or in constraints of a model.

An instance of `IloSolver` or an instance of `IloCP` extracts the object returned by `IloPower`.

Global function operator<=

```
public IloConstraint operator<=(IloNumExprArg base, IloNumExprArg base2)
public IloRange operator<=(IloNumExprArg base, IloNum val)
public IloRangeBase operator<=(IloNum val, const IloNumExprArg expr)
```

Definition file: ilconcert/ilolinear.h

overloaded C++ operator
This overloaded C++ operator constrains its first argument to be less than or equal to its second argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function IloEnableNANDetection

```
public void IloEnableNANDetection()
```

Definition file: ilconcert/ilosys.h

Enables NaN (Not a number) detection.
This function enables your application to detect invalid data as a NaN (Not a number).

Global function IloInitMT

```
public void IloInitMT()
public void IloInitMT(IloBaseEnvMutex *)
```

Definition file: ilconcert/iloenv.h

Initializes multithreading.
This function initializes multithreading in a Concert Technology application.

Global function operator%

```
public IloIntExprArg operator%(IloInt x, const IloIntExprArg y)
```

Definition file: ilconcert/iloexpression.h

Returns an expression equal to the modulo of its arguments.
This operator returns an instance of `IloIntExprArg`, the internal building block of an expression, representing the modulo of the integer value `x` and the expression `y`.

Global function operator%

```
public IloIntExprArg operator%(const IloIntExprArg x, IloInt y)
```

Definition file: ilconcert/iloexpression.h

Returns an expression equal to the modulo of its arguments.
This operator returns an instance of `IloIntExprArg`, the internal building block of an expression, representing the modulo of the expression `x` and the integer value `y`.

Global function IloMaximize

```
public IloObjective IloMaximize(const IloEnv env, IloNum constant=0.0, const char *
name=0)
public IloObjective IloMaximize(const IloEnv env, const IloNumExprArg expr, const
char * name=0)
```

Definition file: ilconcert/ilolinear.h

Defines a maximization objective.
This function defines a maximization objective in a model. In other words, it simply offers a convenient way to create an instance of `IloObjective` with its sense defined as `Maximize`. However, an instance of `IloObjective` created by `IloMaximize` may not necessarily maintain its sense throughout the lifetime of the instance. The optional argument `name` is set to `0` by default.

You may define more than one objective in a model. However, algorithms conventionally take into account only one objective at a time.

Global function IloLog

```
public IloNumExprArg IloLog(const IloNumExprArg arg)
public IloNum IloLog(IloNum val)
```

Definition file: ilconcert/iloexpression.h

Returns the natural logarithm of its argument.
Concert Technology offers predefined functions that return an expression from an algebraic function on expressions. These predefined functions also return a numeric value from an algebraic function on numeric values as well.

`IloLog` returns the natural logarithm of its argument. In order to conform to IEEE 754 standards for floating-point arithmetic, you should use this function in your Concert Technology applications, rather than the standard C++ `log`.

Global function IloDisableNANDetection

```
public void IloDisableNANDetection()
```

Definition file: ilconcert/ilosys.h

Disables NaN (Not a number) detection.
This function turns off NaN (Not a number) detection.

Global function operator<<

```
public ostream & operator<<(ostream & out, const IloExtractable & ext)
```

Definition file: ilconcert/iloextractable.h

Overloaded C++ operator.
This overloaded C++ operator directs output to an output stream.

Global function operator<<

```
public ostream & operator<<(ostream & out, const IloCsvLine & line)
```

Definition file: ilconcert/ilocsvreader.h

Overloaded operator for csv output.
This operator has been overloaded to treat an `IloCsvLine` object appropriately as output. It directs its output to an output stream (normally, standard output) and displays information about its second argument `line`.

Global function operator<<

```
public ostream & operator<<(ostream & out, IloAlgorithm::Status st)
public ostream & operator<<(ostream & out, const IloArray< X > & a)
public ostream & operator<<(ostream & out, const IloNumExpr & ext)
public ostream & operator<<(ostream & os, const IloRandom & r)
public ostream & operator<<(ostream & stream, const IloSolution & solution)
public ostream & operator<<(ostream & stream, const IloSolutionManip & fragment)
public ostream & operator<<(ostream & o, const IloException & e)
```

Definition file: ilconcert/iloalg.h

Overloaded C++ operator.
This overloaded C++ operator directs output to an output stream.

Global function operator-

```
public IloNumToNumStepFunction operator-(const IloNumToNumStepFunction f1, const
IloNumToNumStepFunction f2)
```

Definition file: ilconcert/ilonumfunc.h

Creates and returns a function equal to the difference between its argument functions.
This operator creates and returns a function equal to the difference between functions `f1` and `f2`. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: `IloNumToNumStepFunction`.

Global function operator-

```
public IloNumExprArg operator-(const IloNumExprArg x, const IloNumExprArg y)
public IloNumExprArg operator-(const IloNumExprArg x, IloNum y)
public IloNumExprArg operator-(IloNum x, const IloNumExprArg y)
public IloIntExprArg operator-(const IloIntExprArg x, const IloIntExprArg y)
public IloIntExprArg operator-(const IloIntExprArg x, IloInt y)
public IloIntExprArg operator-(IloInt x, const IloIntExprArg y)
```

Definition file: ilconcert/iloexpression.h

Returns an expression equal to the difference of its arguments.
This overloaded C++ operator returns an expression equal to the difference of its arguments. Its arguments may be numeric values, numeric variables, or other expressions.

Global function IloDiv

```
public IloIntExprArg IloDiv(const IloIntExprArg x, const IloIntExprArg y)
public IloIntExprArg IloDiv(const IloIntExprArg x, IloInt y)
public IloIntExprArg IloDiv(IloInt x, const IloIntExprArg y)
```

Definition file: ilconcert/iloexpression.h

Integer division function.
This function is available for integer division. For numeric division, use operator/.

Global function IloSquare

```
public IloNumExprArg IloSquare(const IloNumExprArg arg)
public IloNum IloSquare(IloNum val)
public IloInt IloSquare(IloInt val)
public IloInt IloSquare(int val)
public IloIntExprArg IloSquare(const IloIntExprArg arg)
```

Definition file: ilconcert/iloexpression.h

Returns the square of its argument.
Concert Technology offers predefined functions that return an expression from an algebraic function over expressions. These predefined functions also return a numeric value from an algebraic function over numeric values as well.

`IloSquare` returns the square of its argument (that is, `val*val` or `expr*expr`).

What Is Extracted

`IloSquare` is extracted by an instance of `IloCplex` as a quadratic term. If the quadratic term is positive semi-definite, it may appear in an objective function or constraint.

`IloSquare` is extracted by an instance of `IloCP` or `IloSolver` as an instance of `IlcSquare`.

Global function IloMinimize

```
public IloObjective IloMinimize(const IloEnv env, IloNum constant=0.0, const char *
name=0)
public IloObjective IloMinimize(const IloEnv env, const IloNumExprArg expr, const
char * name=0)
```

Definition file: ilconcert/ilolinear.h

Defines a minimization objective.
This function defines a minimization objective in a model. In other words, it simply offers a convenient way to create an instance of `IloObjective` with its sense defined as `Minimize`. However, an instance of `IloObjective` created by `IloMinimize` may not necessarily maintain its sense throughout the lifetime of the instance. The optional argument `name` is set to `0` by default.

You may define more than one objective in a model. However, algorithms conventionally take into account only one objective at a time.

Global function IloRound

```
public IloNum IloRound(IloNum val)
```

Definition file: ilconcert/iloenv.h

Computes the nearest integer value to its argument.
This function computes the nearest integer value to `val`. Halfway cases are rounded to the larger in magnitude.

Examples:

```
IloRound(IloInfinity) is IloInfinity.
IloRound(-IloInfinity) is -IloInfinity.
IloRound(0) is 0.
IloRound(0.4) is 0.
IloRound(-0.4) is 0.
IloRound(0.5) is 1.
IloRound(-0.5) is -1.
IloRound(0.6) is 1.
IloRound(-0.6) is -1.
IloRound(1e300) is 1e300.
IloRound(1.0000001e6) is 1e6.
IloRound(1.0000005e6) is 1.000001e6.
```

Global function operator+

```
public IloNumToNumStepFunction operator+(const IloNumToNumStepFunction f1, const
IloNumToNumStepFunction f2)
```

Definition file: ilconcert/ilonumfunc.h

Creates and returns a function equal the sum of its argument functions.
This operator creates and returns a function equal the sum of the functions `f1` and `f2`. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: `IloNumToNumStepFunction`.

Global function operator+

```
public IloNumExprArg operator+(const IloNumExprArg x, const IloNumExprArg y)
public IloNumExprArg operator+(const IloNumExprArg x, IloNum y)
public IloNumExprArg operator+(IloNum x, const IloNumExprArg y)
public IloIntExprArg operator+(const IloIntExprArg x, const IloIntExprArg y)
public IloIntExprArg operator+(const IloIntExprArg x, IloInt y)
public IloIntExprArg operator+(IloInt x, const IloIntExprArg y)
```

Definition file: ilconcert/iloexpression.h

Returns an expression equal to the sum of its arguments.
This overloaded C++ operator returns an expression equal to the sum of its arguments. Its arguments may be numeric values, numeric variables, or other expressions.

Global function IloIsNAN

```
public int IloIsNAN(double)
```

Definition file: ilconcert/ilosys.h

Tests whether a double value is a NaN.
This function tests whether a double value is a NaN (Not a number).

Global function IloSum

```
public IloNumExprArg IloSum(const IloNumExprArray exprs)
public IloIntExprArg IloSum(const IloIntExprArray exprs)
public IloNumExprArg IloSum(const IloNumVarArray exprs)
public IloIntExprArg IloSum(const IloIntVarArray exprs)
public IloNum IloSum(const IloNumArray values)
public IloInt IloSum(const IloIntArray values)
```

Definition file: ilconcert/iloexpression.h

Returns a numeric value representing the sum of numeric values.
These functions return a numeric value representing the sum of numeric values in the array `vals`, or an instance of `IloNumExprArg`, the internal building block of an expression, representing the sum of the variables in the arrays `exprs` or `values`.

Global function IloExponent

```
public IloNumExprArg IloExponent(const IloNumExprArg arg)
public IloNum IloExponent(IloNum val)
```

Definition file: ilconcert/iloexpression.h

Returns the exponent of its argument.
Concert Technology offers predefined functions that return an expression from an algebraic function on expressions. These predefined functions also return a numeric value from an algebraic function on numeric values as well.

`IloExponent` returns the exponentiation of its argument. In order to conform to IEEE 754 standards for floating-point arithmetic, you should use this function in your Concert Technology applications, rather than the standard C++ `exp`.

Global function operator new

```
public void * operator new(size_t sz, const IloEnv & env)
```

Definition file: ilconcert/iloenv.h

Overloaded C++ `new` operator.
IBM ILOG Concert Technology offers this overloaded C++ `new` operator. This operator is overloaded to allocate data on internal data structures associated with an invoking environment (an instance of `IloEnv`). The memory used by objects allocated with this overloaded operator is automatically reclaimed when you call the member function `IloEnv::end`. As a developer, you must *not* delete objects allocated with this operator because of this automatic freeing of memory.

In other words, you must *not* use the `delete` operator for objects allocated with this overloaded `new` operator.

The use of this overloaded `new` operator is not obligatory in Concert Technology applications. You will see examples of its use in the user's manuals that accompany the IBM ILOG optimization products.

Global function operator&&

```
public IloAnd operator&&(const IloConstraint constraint1, const IloConstraint
constraint2)
```

Definition file: ilconcert/ilomodel.h

Overloaded C++ operator for conjunctive constraints.
This overloaded C++ operator creates a conjunctive constraint that represents the conjunction of its two arguments. The constraint can represent a conjunction of two constraints; of a constraint and another conjunction; or of two conjunctions. In order to be taken into account, this constraint must be added to a model and extracted by an algorithm, such as `IloCplex` or `IloSolver`.

Global function IloGetClone

```
public X IloGetClone(IloEnvI * env, const X x)
```

Definition file: ilconcert/iloextractable.h

Creates a clone.
This C++ template creates a clone (that is, an exact copy) of an instance of the class X.

Global function operator<

```
public IloConstraint operator<(IloNumExprArg base, IloNumExprArg base2)
public IloConstraint operator<(IloNumExprArg base, IloNum val)
public IloConstraint operator<(IloNum val, const IloNumExprArg expr)
public IloConstraint operator<(IloIntExprArg base, IloIntExprArg base2)
public IloConstraint operator<(IloIntExprArg base, IloInt val)
```

Definition file: ilconcert/ilolinear.h

overloaded C++ operator
This overloaded C++ operator constrains its first argument to be strictly less than its second argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function IloLexicographic

```
public IloConstraint IloLexicographic(IloEnv env, IloIntExprArray x,
IloIntExprArray y, const char *=0)
```

Definition file: ilconcert/ilomodel.h

Returns a constraint which maintains two arrays to be lexicographically ordered.
The `IloLexicographic` function returns a constraint which maintains two arrays to be lexicographically ordered.

More specifically, `IloLexicographic(x, y)` maintains that `x` is less than or equal to `y` in the lexicographical sense of the term. This means that either both arrays are equal or that there exists `i < size(x)` such that for all `j < i`, `x[j] = y[j]` and `x[i] < y[i]`.

Note that the size of the two arrays must be the same.

Global function IloMax

```
public IloNumToNumStepFunction IloMax(const IloNumToNumStepFunction f1, const
IloNumToNumStepFunction f2)
```

Definition file: ilconcert/ilonumfunc.h

Creates and returns a function equal to the maximal value of its argument functions.
This operator creates and returns a function equal to the maximal value of the functions `f1` and `f2`. That is, for all points `x` in the definition interval, the resulting function is equal to the `max(f1(x), f2(x))`. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: `IloNumToNumStepFunction`.

Global function IloMax

```
public IloNum IloMax(const IloNumArray vals)
public IloNum IloMax(IloNum val1, IloNum val2)
public IloInt IloMax(const IloIntArray vals)
public IloNumExprArg IloMax(const IloNumExprArray exprs)
public IloIntExprArg IloMax(const IloIntExprArray exprs)
public IloNumExprArg IloMax(const IloNumExprArg x, const IloNumExprArg y)
public IloNumExprArg IloMax(const IloNumExprArg x, IloNum y)
public IloNumExprArg IloMax(IloNum x, const IloNumExprArg y)
public IloIntExprArg IloMax(const IloIntExprArg x, const IloIntExprArg y)
public IloIntExprArg IloMax(const IloIntExprArg x, IloInt y)
public IloNumExprArg IloMax(const IloIntExprArg x, IloNum y)
public IloIntExprArg IloMax(const IloIntExprArg x, int y)
public IloIntExprArg IloMax(IloInt x, const IloIntExprArg y)
public IloNumExprArg IloMax(IloNum x, const IloIntExprArg y)
public IloIntExprArg IloMax(int x, const IloIntExprArg y)
```

Definition file: ilconcert/iloexpression.h

Returns a numeric value representing the max of numeric values.
These functions compare their arguments and return the greatest value.

Global function operator||

```
public IloOr operator||(const IloConstraint constraint1, const IloConstraint
constraint2)
```

Definition file: ilconcert/ilomodel.h

Overloaded C++ operator for a disjunctive constraint.
This overloaded C++ operator creates a disjunctive constraint that represents the disjunction of its two arguments. The constraint can represent a disjunction of two constraints; of a constraint and another disjunction; or of two disjunctions. In order to be taken into account, this constraint must be added to a model and extracted by an algorithm, such as `IloCplex` or `IloSolver`.

Global function operator*

```
public IloNumLinExprTerm operator*(const IloNumVar x, IloInt num)
public IloNumLinExprTerm operator*(IloInt num, const IloNumVar x)
public IloNumLinExprTerm operator*(const IloIntVar x, IloNum num)
public IloNumLinExprTerm operator*(IloNum num, const IloIntVar x)
public IloIntLinExprTerm operator*(const IloIntVar x, IloInt num)
public IloNumExprArg operator*(const IloNumExprArg x, const IloNumExprArg y)
public IloNumExprArg operator*(const IloNumExprArg x, IloNum y)
public IloNumExprArg operator*(IloNum x, const IloNumExprArg y)
public IloIntExprArg operator*(const IloIntExprArg x, const IloIntExprArg y)
public IloIntExprArg operator*(const IloIntExprArg x, IloInt y)
```

Definition file: ilconcert/iloexpression.h

Returns an expression equal to the product of its arguments.
This overloaded C++ operator returns an expression equal to the product of its arguments. Its arguments may be numeric values, numeric variables, or other expressions.

Global function operator*

```
public IloNumToNumStepFunction operator*(const IloNumToNumStepFunction f1, IloNum
k)
public IloNumToNumStepFunction operator*(IloNum k, const IloNumToNumStepFunction
f1)
```

Definition file: ilconcert/ilonumfunc.h

Creates and returns a function equal to its argument function multiplied by a given factor.
These operators create and return a function equal to the function `f1` multiplied by a factor `k` everywhere on the definition interval. The resulting function is defined on the same interval as the argument function `f1`. See also: `IloNumToNumStepFunction`.

Global function operator==

```
public IloConstraint operator==(IloNumExprArg base, IloNumExprArg expr)
public IloRange operator==(IloNumExprArg base, IloNum val)
public IloRange operator==(IloNum val, IloNumExprArg eb)
```

Definition file: ilconcert/ilolinear.h

Overloaded C++ operator.
This overloaded C++ operator constrains its two arguments to be equal. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function operator!

```
public IloConstraint operator!(const IloConstraint constraint)
```

Definition file: ilconcert/ilomodel.h

Overloaded C++ operator for negation.
This overloaded C++ operator returns a constraint that is the negation of its argument. In order to be taken into account, this constraint must be added to a model and extracted by an algorithm, such as `IloCplex` or `IloSolver`.

Global function IloMin

```
public IloNumToNumStepFunction IloMin(const IloNumToNumStepFunction f1, const
IloNumToNumStepFunction f2)
```

Definition file: ilconcert/ilonumfunc.h

Creates and returns a function equal to the minimal value of its argument functions.
This operator creates and returns a function equal to the minimal value of the functions `f1` and `f2`. That is, for all points `x` in the definition interval, the resulting function is equal to the `min(f1(x), f2(x))`. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: `IloNumToNumStepFunction`.

Global function IloMin

```
public IloNum IloMin(const IloNumArray vals)
public IloNum IloMin(IloNum val1, IloNum val2)
public IloInt IloMin(const IloIntArray vals)
public IloNumExprArg IloMin(const IloNumExprArray exprs)
public IloIntExprArg IloMin(const IloIntExprArray exprs)
public IloNumExprArg IloMin(const IloNumExprArg x, const IloNumExprArg y)
public IloNumExprArg IloMin(const IloNumExprArg x, IloNum y)
public IloNumExprArg IloMin(IloNum x, const IloNumExprArg y)
public IloIntExprArg IloMin(const IloIntExprArg x, const IloIntExprArg y)
public IloIntExprArg IloMin(const IloIntExprArg x, IloInt y)
public IloNumExprArg IloMin(const IloIntExprArg x, IloNum y)
public IloIntExprArg IloMin(const IloIntExprArg x, int y)
public IloIntExprArg IloMin(IloInt x, const IloIntExprArg y)
public IloNumExprArg IloMin(IloNum x, const IloIntExprArg y)
public IloIntExprArg IloMin(int x, const IloIntExprArg y)
```

Definition file: ilconcert/iloexpression.h

Returns a numeric value representing the min of numeric values.
These functions compare their arguments and return the least value. When its argument is an array, the function compares the *elements* of that array and returns the least value.

Global function IloAdd

```
public X IloAdd(IloModel & mdl, X x)
```

Definition file: ilconcert/ilomodel.h

Template to add elements to a model.
This C++ template helps when you want to add elements to a model. In those synopses, `X` represents a class, `x` is an instance of the class `X`. The class `X` must be `IloExtractable`, `IloExtractableArray`, or one of their subclasses.

If `model` is an instance of `IloModel`, derived from `IloExtractable`, then `x` will be added to the top level of that model.

As an alternative to this way of adding extractable objects to a model, you may also use `IloModel::add`.

This template preserves the original type of its argument `x` when it returns `x`. This feature of the template may be useful, for example, in cases like this:

```
IloRange rng = IloAdd(model, 3 * x + y == 17);
```

For a comparison of these two ways of adding extractable objects to a model, see Adding Extractable Objects in the documentation of `IloExtractable`.

See Also: IloAnd, IloExtractable, IloExtractableArray, IloModel, IloOr

Global function IloScalProd

```
public IloNumExprArg IloScalProd(const IloNumArray values, const IloNumVarArray
vars)
public IloNumExprArg IloScalProd(const IloNumVarArray vars, const IloNumArray
values)
public IloNumExprArg IloScalProd(const IloNumArray values, const IloIntVarArray
vars)
public IloNumExprArg IloScalProd(const IloIntVarArray vars, const IloNumArray
values)
public IloNumExprArg IloScalProd(const IloIntArray values, const IloNumVarArray
vars)
public IloNumExprArg IloScalProd(const IloNumVarArray vars, const IloIntArray
values)
public IloNumExprArg IloScalProd(const IloNumExprArray leftExprs, const
IloNumExprArray rightExprs)
```

Definition file: ilconcert/iloexpression.h

Represents the scalar product.
This function returns an instance of `IloNumExprArg`, the internal building block of an expression, representing the scalar product of the variables in the arrays `vars` and `values` or the arrays `leftExprs` and `rightExprs`.

Global function IloScalProd

```
public IloNum IloScalProd(const IloNumArray vals1, const IloNumArray vals2)
public IloNum IloScalProd(const IloIntArray vals1, const IloNumArray vals2)
public IloNum IloScalProd(const IloNumArray vals1, const IloIntArray vals2)
```

Definition file: ilconcert/iloexpression.h

Represents the scalar product.
This function returns a numeric value representing the scalar product of numeric values in the arrays `vals1` and `vals2`.

Global function IloScalProd

```
public IloIntExprArg IloScalProd(const IloIntArray values, const IloIntVarArray
vars)
public IloIntExprArg IloScalProd(const IloIntVarArray vars, const IloIntArray
values)
public IloIntExprArg IloScalProd(const IloIntExprArray leftExprs, const
IloIntExprArray rightExprs)
```

Definition file: ilconcert/iloexpression.h

Represents the scalar product.
This function returns an instance of `IloIntExprArg`, the internal building block of an integer expression, representing the scalar product of the variables in the arrays `vars` and `values` or the arrays `leftExprs` and `rightExprs`.

Global function IloScalProd

```
public IloInt IloScalProd(const IloIntArray vals1, const IloIntArray vals2)
```

Definition file: ilconcert/iloexpression.h

Represents the scalar product.
This function returns an integer value representing the scalar product of integer values in the arrays `vals1` and `vals2`.

Macro ILOBARRIERCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOBARRIERCALLBACK0(name)
ILOBARRIERCALLBACK1(name, type1, x1)
ILOBARRIERCALLBACK2(name, type1, x1, type2, x2)
ILOBARRIERCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOBARRIERCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOBARRIERCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOBARRIERCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOBARRIERCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOBARRIERCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::BarrierCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::BarrierCallbackI

Macro ILOBRANCHCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOBRANCHCALLBACK0(name)
ILOBRANCHCALLBACK1(name, type1, x1)
ILOBRANCHCALLBACK2(name, type1, x1, type2, x2)
ILOBRANCHCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOBRANCHCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOBRANCHCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOBRANCHCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOBRANCHCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOBRANCHCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::BranchCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::BranchCallbackI

Macro ILOCONTINUOUSCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOCONTINUOUSCALLBACK0(name)
ILOCONTINUOUSCALLBACK1(name, type1, x1)
ILOCONTINUOUSCALLBACK2(name, type1, x1, type2, x2)
ILOCONTINUOUSCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOCONTINUOUSCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOCONTINUOUSCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOCONTINUOUSCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOCONTINUOUSCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOCONTINUOUSCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::ContinuousCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::ContinuousCallbackI

Macro ILOCPLEXGOAL0

Definition file: ilcplex/ilogoals.h

```
ILOCPLEXGOAL0(name)
ILOCPLEXGOAL1(name, type0, var0)
ILOCPLEXGOAL2(name, type0, var0, type1, var1)
ILOCPLEXGOAL3(name, type0, var0, type1, var1, type2, var2)
ILOCPLEXGOAL4(name, t0, v0, t1, v1, t2, v2, t3, v3)
ILOCPLEXGOAL5(name, t0, v0, t1, v1, t2, v2, t3, v3, t4, v4)
ILOCPLEXGOAL6(name, t0, v0, t1, v1, t2, v2, t3, v3, t4, v4, t5, v5)
```

This macro defines a user goal class named `nameI` and a constructor named `name` with *n* data members, where *n* is the number following `ILOCPLEXGOAL`. The first parameter of this macro is always the name of the constructor to be created. What follows are *n* pairs of parameters, each parameter specifying a data member of the goal. The first parameter of such a pair specifies the type of the data member and is denoted as `Ti` in the macro definition above. The second parameter of such a pair, denoted by `datai`, specifies the data member's name.

The constructor `name` created by this function will have `IloEnvenv` as its first argument, followed by *n* additional arguments. The constructor creates a new instance of the user-written goal class `nameI` and populates its data members with the arguments following `IloEnvenv` in the function argument list. The constructor `name` is what you should use to create new goal objects.

The call to the macro must be followed immediately by the `execute` method of the goal class. This method must be enclosed in curly brackets, as shown in the examples that follow. The macro will also generate an implementation of the method `duplicateGoal` that simply calls the default constructor for the new class `nameI`.

You are not obliged to use this macro to define goals. In particular, if your data members do not permit the use of the default constructor as an implementation of the method `duplicateGoal` or the default destructor, you must subclass `IloCplex::GoalI` directly and implement those methods appropriately.

Since the argument `name` is used to construct the name of the goal's implementation class, it is not possible to use the same name for several goal definitions.

Example

Here's how to define a goal with one data member:

```
    ILOCPLEXGOAL1(PrintX, IloInt, x) {
      IloEnv env = getEnv();
      env.out() << "PrintX: a goal with one data member" <<
endl;
      env.out() << x << endl;
      return 0;
    }
```

This macro generates code similar to the following lines:

```
    class PrintXI : public IloCplex::GoalI {
    public:
      IloInt x;
      PrintXI(IloEnv env, IloInt arg1)
      IloCplex::Goal execute();
      IloCplex::Goal duplicateGoal();
    };

    PrintXI::PrintXI(IloEnv env, IloInt arg1) :
                                  IloCplex::GoalI(env),
                                  x(arg1) {
    }

    IloCplex::Goal PrintX(IloEnv env, IloInt x) {
      return new PrintXI(env, x);
    }
```

```
    IloCplex::Goal PrintXI::execute() {
      IloEnv env = getEnv();
      env.out() << "PrintX: a goal with one data member" <<
endl;
      env.out() << x << endl;
      return 0;
    }

    IloCplex::Goal PrintXI::duplicateGoal() {
      return new PrintXI(getEnv(), x);
    }
```

Macro ILOCROSSOVERCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOCROSSOVERCALLBACK0(name)
ILOCROSSOVERCALLBACK1(name, type1, x1)
ILOCROSSOVERCALLBACK2(name, type1, x1, type2, x2)
ILOCROSSOVERCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOCROSSOVERCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOCROSSOVERCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOCROSSOVERCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOCROSSOVERCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOCROSSOVERCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::CrossoverCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::CallbackI, IloCplex::CrossoverCallbackI

Macro ILOCUTCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOCUTCALLBACK0(name)
ILOCUTCALLBACK1(name, type1, x1)
ILOCUTCALLBACK2(name, type1, x1, type2, x2)
ILOCUTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOCUTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOCUTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOCUTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6,
x6)
ILOCUTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6,
x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle to it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOCUTCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::CutCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::CutCallbackI

Macro ILODISJUNCTIVECUTCALLBACK0

Definition file: ilcplex/ilocplex.h

ILODISJUNCTIVECUTCALLBACK0(name)
ILODISJUNCTIVECUTCALLBACK1(name, type1, x1)
ILODISJUNCTIVECUTCALLBACK2(name, type1, x1, type2, x2)
ILODISJUNCTIVECUTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILODISJUNCTIVECUTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILODISJUNCTIVECUTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILODISJUNCTIVECUTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)
ILODISJUNCTIVECUTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle to it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILODISJUNCTIVECUTCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::DisjunctiveCutCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::DisjunctiveCutCallbackI

Macro ILODISJUNCTIVECUTINFOCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILODISJUNCTIVECUTINFOCALLBACK0(name)
ILODISJUNCTIVECUTINFOCALLBACK1(name, type1, x1)
ILODISJUNCTIVECUTINFOCALLBACK2(name, type1, x1, type2, x2)
ILODISJUNCTIVECUTINFOCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILODISJUNCTIVECUTINFOCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILODISJUNCTIVECUTINFOCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5)
ILODISJUNCTIVECUTINFOCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6)
ILODISJUNCTIVECUTINFOCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle to it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILODISJUNCTIVECUTINFOCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::DisjunctiveCutInfoCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::DisjunctiveCutInfoCallbackI

Macro ILOFLOWMIRCUTCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOFLOWMIRCUTCALLBACK0(name)
ILOFLOWMIRCUTCALLBACK1(name, type1, x1)
ILOFLOWMIRCUTCALLBACK2(name, type1, x1, type2, x2)
ILOFLOWMIRCUTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOFLOWMIRCUTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOFLOWMIRCUTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOFLOWMIRCUTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOFLOWMIRCUTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first argument, followed by the n arguments specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOFLOWMIRCUTCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::FlowMIRCutCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, the team recommends that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::FlowMIRCutCallbackI

Macro ILOFLOWMIRCUTINFOCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOFLOWMIRCUTINFOCALLBACK0(name)
ILOFLOWMIRCUTINFOCALLBACK1(name, type1, x1)
ILOFLOWMIRCUTINFOCALLBACK2(name, type1, x1, type2, x2)
ILOFLOWMIRCUTINFOCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOFLOWMIRCUTINFOCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOFLOWMIRCUTINFOCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5)
ILOFLOWMIRCUTINFOCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6)
ILOFLOWMIRCUTINFOCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first argument, followed by the n arguments specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOFLOWMIRINFOCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::FlowMIRCutInfoCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, the team recommends that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::FlowMIRCutInfoCallbackI

Macro ILOFRACTIONALCUTCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOFRACTIONALCUTCALLBACK0(name)
ILOFRACTIONALCUTCALLBACK1(name, type1, x1)
ILOFRACTIONALCUTCALLBACK2(name, type1, x1, type2, x2)
ILOFRACTIONALCUTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOFRACTIONALCUTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOFRACTIONALCUTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5)
ILOFRACTIONALCUTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6)
ILOFRACTIONALCUTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOFRACTIONALCUTCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::FractionalCutCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::FractionalCutCallbackI

Macro ILOFRACTIONALCUTINFOCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOFRACTIONALCUTINFOCALLBACK0(name)
ILOFRACTIONALCUTINFOCALLBACK1(name, type1, x1)
ILOFRACTIONALCUTINFOCALLBACK2(name, type1, x1, type2, x2)
ILOFRACTIONALCUTINFOCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOFRACTIONALCUTINFOCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOFRACTIONALCUTINFOCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5)
ILOFRACTIONALCUTINFOCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6)
ILOFRACTIONALCUTINFOCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4,
type5, x5, type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOFRACTIONALCUTINFOCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::FractionalCutInfoCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::FractionalCutInfoCallbackI

Macro IloHalfPi

Definition file: ilconcert/ilosys.h

`IloHalfPi`

Half pi.
Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloHalfPi;       // = 1.5707963267948966192
```

Macro ILOHEURISTICCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOHEURISTICCALLBACK0(name)
ILOHEURISTICCALLBACK1(name, type1, x1)
ILOHEURISTICCALLBACK2(name, type1, x1, type2, x2)
ILOHEURISTICCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOHEURISTICCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOHEURISTICCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOHEURISTICCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOHEURISTICCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOHEURISTICCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::HeuristicCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::HeuristicCallbackI

Macro ILOINCUMBENTCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOINCUMBENTCALLBACK0(name)
ILOINCUMBENTCALLBACK1(name, type1, x1)
ILOINCUMBENTCALLBACK2(name, type1, x1, type2, x2)
ILOINCUMBENTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOINCUMBENTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOINCUMBENTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOINCUMBENTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOINCUMBENTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOINCUMBENTCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::IncumbentCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::IncumbentCallbackI

Macro ILOLAZYCONSTRAINTCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOLAZYCONSTRAINTCALLBACK0(name)
ILOLAZYCONSTRAINTCALLBACK1(name, type1, x1)
ILOLAZYCONSTRAINTCALLBACK2(name, type1, x1, type2, x2)
ILOLAZYCONSTRAINTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOLAZYCONSTRAINTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOLAZYCONSTRAINTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5)
ILOLAZYCONSTRAINTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6)
ILOLAZYCONSTRAINTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined lazy constraint callback named `nameI` and a function named `name` that creates an instance of this class and returns an `IloCplex::Callback` handle for it. This function needs to be called with an environment as first parameter followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of method `makeClone` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOLAZYCONSTRAINTCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::LazyConstraintCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::LazyConstraintCallbackI

Macro ILOMIPCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOMIPCALLBACK0(name)
ILOMIPCALLBACK1(name, type1, x1)
ILOMIPCALLBACK2(name, type1, x1, type2, x2)
ILOMIPCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOMIPCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOMIPCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOMIPCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6,
x6)
ILOMIPCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6,
x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOMIPCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::MIPCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::MIPCallbackI

Macro ILOMIPINFOCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOMIPINFOCALLBACK0(name)
ILOMIPINFOCALLBACK1(name, type1, x1)
ILOMIPINFOCALLBACK2(name, type1, x1, type2, x2)
ILOMIPINFOCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOMIPINFOCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOMIPINFOCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOMIPINFOCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOMIPINFOCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOMIPINFOCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::MIPInfoCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::MIPInfoCallbackI

Macro ILONETWORKCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILONETWORKCALLBACK0(name)
ILONETWORKCALLBACK1(name, type1, x1)
ILONETWORKCALLBACK2(name, type1, x1, type2, x2)
ILONETWORKCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILONETWORKCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILONETWORKCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILONETWORKCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILONETWORKCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILONETWORKCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::NetworkCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::NetworkCallbackI

Macro ILONODECALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILONODECALLBACK0(name)
ILONODECALLBACK1(name, type1, x1)
ILONODECALLBACK2(name, type1, x1, type2, x2)
ILONODECALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILONODECALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILONODECALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILONODECALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILONODECALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILONODECALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::NodeCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::NodeCallbackI

Macro IloPi

Definition file: ilconcert/ilosys.h

`IloPi`

Pi.
Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloPi;           // = 3.1415926535897932846
```

Macro ILOPRESOLVECALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOPRESOLVECALLBACK0(name)
ILOPRESOLVECALLBACK1(name, type1, x1)
ILOPRESOLVECALLBACK2(name, type1, x1, type2, x2)
ILOPRESOLVECALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOPRESOLVECALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOPRESOLVECALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOPRESOLVECALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOPRESOLVECALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOPRESOLVECALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::PresolveCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::PresolveCallbackI

Macro ILOPROBINGCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOPROBINGCALLBACK0(name)
ILOPROBINGCALLBACK1(name, type1, x1)
ILOPROBINGCALLBACK2(name, type1, x1, type2, x2)
ILOPROBINGCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOPROBINGCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOPROBINGCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOPROBINGCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOPROBINGCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOPROBINGCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::ProbingCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::ProbingCallbackI

Macro ILOPROBINGINFOCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOPROBINGINFOCALLBACK0(name)
ILOPROBINGINFOCALLBACK1(name, type1, x1)
ILOPROBINGINFOCALLBACK2(name, type1, x1, type2, x2)
ILOPROBINGINFOCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOPROBINGINFOCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOPROBINGINFOCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5)
ILOPROBINGINFOCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6)
ILOPROBINGINFOCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5,
x5, type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOPROBINGINFOCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::ProbingInfoCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::ProbingInfoCallbackI

Macro IloQuarterPi

Definition file: ilconcert/ilosys.h

`IloQuarterPi`

Quarter pi.
Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloQuarterPi;    // = 0.7853981633974830962
```

Macro ILOSIMPLEXCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOSIMPLEXCALLBACK0(name)
ILOSIMPLEXCALLBACK1(name, type1, x1)
ILOSIMPLEXCALLBACK2(name, type1, x1, type2, x2)
ILOSIMPLEXCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOSIMPLEXCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOSIMPLEXCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOSIMPLEXCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOSIMPLEXCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOSIMPLEXCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::SimplexCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::SimplexCallbackI

Macro ILOSOLVECALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOSOLVECALLBACK0(name)
ILOSOLVECALLBACK1(name, type1, x1)
ILOSOLVECALLBACK2(name, type1, x1, type2, x2)
ILOSOLVECALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOSOLVECALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOSOLVECALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOSOLVECALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOSOLVECALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOSOLVECALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::SolveCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::SolveCallbackI

Macro ILOSTLBEGIN

Definition file: ilconcert/ilosys.h

ILOSTLBEGIN

Macro for STL.
This macro enables you run your application either with the STL (Standard Template Library) of Microsoft Visual C++ or with other platforms. It is defined as:

```
using namespace std
```

when the STL is used (ports of type `stat_sta`, `stat_mta` or `stat_mda`); otherwise, its value is simply null.

Macro IloThreeHalfPi

Definition file: ilconcert/ilosys.h

`IloThreeHalfPi`

Three half-pi.
Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloThreeHalfPi;  // = 4.71238898038468985769
```

Macro ILOTUNINGCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOTUNINGCALLBACK0(name)
ILOTUNINGCALLBACK1(name, type1, x1)
ILOTUNINGCALLBACK2(name, type1, x1, type2, x2)
ILOTUNINGCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOTUNINGCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOTUNINGCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOTUNINGCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOTUNINGCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first argument, followed by the n arguments specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOTUNINGCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::TuningCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, the team recommends that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::TuningCallbackI

Macro IloTwoPi

Definition file: ilconcert/ilosys.h

`IloTwoPi`

Two pi.
Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloTwoPi;        // = 6.2831853071795864 7692
```

Macro ILOUSERCUTCALLBACK0

Definition file: ilcplex/ilocplex.h

```
ILOUSERCUTCALLBACK0(name)
ILOUSERCUTCALLBACK1(name, type1, x1)
ILOUSERCUTCALLBACK2(name, type1, x1, type2, x2)
ILOUSERCUTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOUSERCUTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOUSERCUTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOUSERCUTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6)
ILOUSERCUTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5,
type6, x6, type7, x7)
```

This macro creates two things: an implementation class for a user-defined user cut callback named `nameI` and a function named `name` that creates an instance of this class and returns an `IloCplex::Callback` handle for it. This function needs to be called with an environment as first parameter followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `makeClone` as required for callbacks. The implementation of the `main` method must be provided in curly brackets {} by the user and must follow the macro invocation, like this:

```
ILOUSERCUTCALLBACKn(name, ...) {

// implementation of the callback

}
```

For the implementation of the callback, methods from the class `IloCplex::UserCutCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also: IloCplex::UserCutCallbackI

Typedef IntegerFeasibilityArray

Definition file: ilcplex/ilocplexi.h

```
IloArray< IntegerFeasibility > IntegerFeasibilityArray
```

This type defines an array-type for `IloCplex::ControlCallbackI::IntegerFeasibility`. The fully qualified name of an integer feasibility array is `IloCplex::ControlCallbackI::IntegerFeasibility::Array`.

See Also: IloCplex, IloCplex::ControlCallbackI, ControlCallbackI::IntegerFeasibility

Typedef IntegerFeasibilityArray

Definition file: ilcplex/ilocplexi.h

```
IloArray< IntegerFeasibility > IntegerFeasibilityArray
```

This type defines an array type for `IloCplex::GoalI::IntegerFeasibility`. The fully qualified name of an integer feasibility array is `IloCplex::GoalI::IntegerFeasibilityArray`.

See Also: IloCplex, GoalI::IntegerFeasibility

Typedef IloBool

Definition file: ilconcert/ilosys.h

```
IloInt IloBool
```

Type for Boolean values.
This type definition represents Boolean values in Concert Technology. Those values are `IloTrue` and `IloFalse`. Booleans are, in fact, integers of type `IloInt`. `IloFalse` is 0 (zero), and `IloTrue` is 1 (one). This type anticipates the built-in `bool` type proposed for standard C++. By using this type, you can be sure that the Concert Technology components of your application will port in this respect without source changes across different hardware platforms.

See Also: IloBoolArray, IloInt, IloModel, IloNum

Typedef BasisStatusArray

Definition file: ilcplex/ilocplexi.h

```
IloArray< BasisStatus > BasisStatusArray
```

This type defines an array-type for `IloCplex::BasisStatus`. The fully qualified name of a basis status array is `IloCplex::BasisStatusArray`.

See Also: IloCplex, IloCplex::BasisStatus

Typedef BranchDirectionArray

Definition file: ilcplex/ilocplexi.h

```
IloArray< BranchDirection > BranchDirectionArray
```

This type defines an array-type for `IloCplex::BranchDirection`. The fully qualified name of a branch direction array is `IloCplex::BranchDirectionArray`.

See Also: IloCplex, IloCplex::BranchDirection

Typedef ConflictStatusArray

Definition file: ilcplex/ilocplexi.h

```
IloArray< ConflictStatus > ConflictStatusArray
```

This type defines an array-type for `IloCplex::ConflictStatus`.

See Also: IloCplex, IloCplex::ConflictStatus

Typedef Status

Definition file: ilcplex/ilocplexi.h

```
CplexStatus Status
```

An enumeration for the class `IloAlgorithm`.
`IloAlgorithm` is the base class of algorithms in Concert Technology, and `IloAlgorithm::Status` is an enumeration limited in scope to the class `IloAlgorithm`. The member function `IloAlgorithm::getStatus` returns a status showing information about the current model and the solution.

`Unknown` specifies that the algorithm has no information about the solution of the model.

`Feasible` specifies that the algorithm found a feasible solution (that is, an assignment of values to variables that satisfies the constraints of the model, though it may not necessarily be optimal). The member functions `IloAlgorithm::getValue` access this feasible solution.

`Optimal` specifies that the algorithm found an optimal solution (that is, an assignment of values to variables that satisfies all the constraints of the model and that is proved optimal with respect to the objective of the model). The member functions `IloAlgorithm::getValue` access this optimal solution.

`Infeasible` specifies that the algorithm proved the model infeasible; that is, it is not possible to find an assignment of values to variables satisfying all the constraints in the model.

`Unbounded` specifies that the algorithm proved the model unbounded.

`InfeasibleOrUnbounded` specifies that the model is infeasible or unbounded.

`Error` specifies that an error occurred and, on platforms that support exceptions, that an exception has been thrown.

See Also: IloAlgorithm, operator<<

Typedef IloInt

Definition file: ilconcert/ilosys.h

```
long IloInt
```

Type for signed integers.
This type definition represents signed integers in Concert Technology.

See Also: IloBool, IloModel, IloNum

Typedef IloNum

Definition file: ilconcert/ilosys.h

```
double IloNum
```

Type for numeric values as floating-point numbers.
This type definition represents numeric values as floating-point numbers in Concert Technology.

See Also: IloModel, IloInt

Typedef IloSolutionArray

Definition file: ilconcert/ilosolution.h

```
IloSimpleArray< IloSolution > IloSolutionArray
```

Type definition for arrays of `IloSolution` instances.
This type definition represents arrays of instances of `IloSolution`.

Instances of `IloSolutionArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.

See Also: IloSolution

Variable ILO_NO_MEMORY_MANAGER

Definition file: ilconcert/ilosys.h
OS environment variable controls Concert Technology memory manager.
This operating-system environment variable enables you to control the memory manager of Concert Technology.

Concert Technology uses its own memory manager to provide faster memory allocation for certain Concert Technology objects. The use of this memory manager can hide memory problems normally detected by memory usage applications (such as Rational Purify, for example). If you are working in a software development environment capable of detecting bad memory access, you can use this operating-system environment variable to turn off the Concert Technology memory manager in order to detect such anomalies during software development.

For example, if you are working in such a development environment on a personal computer running Microsoft XP, use this statement:

```
set ILO_NO_MEMORY_MANAGER=1
```

If you are working on a UNIX platform, using a C-shell, here is one way of setting this environment variable:

```
setenv ILO_NO_MEMORY_MANAGER
```

Variable IloInfinity

Definition file: ilconcert/ilosys.h
Largest double-precision floating-point number.
This symbolic constant represents the largest double-precision floating-point number on a given platform. It is initialized when you create an instance of `IloEnv`. In practice, when you use this symbolic constant as an upper bound of a variable in your model, you are effectively stating that the variable is unbounded.

See the IBM ILOG Solver Reference Manual and User's Manual for details about how IBM ILOG Solver treats floating-point calculations in instances of `IloSolver` in conformity with IEEE 754. In particular, IBM ILOG Solver offers other symbolic constants, such as `IlcIntMax` or `IlcFloatMax` that may be more appropriate for your application if you do not intend to state that your variables are effectively unbounded.

See the IBM ILOG CPLEX Reference Manual and User's Manual for details about how IBM ILOG CPLEX treats floating-point calculations in instances of `IloCplex`.

See Also: IloEnv

Variable IloIntMax

Definition file: ilconcert/ilosys.h
Largest integer.
These symbolic constants represent the largest integer on a given platform.

Variable IloIntMin

Definition file: ilconcert/ilosys.h
Least integer.
These symbolic constants represent the least integer on a given platform.