# Using Clearmake in C++ Development

## October 11, 1995 -- Carol Yutkowitz

# C++ templates

## A *template* is a C++ language feature which allows the parameterization of a *container* class (or function).

Example template declaration file (*stack.h*):

```
template<class T, int size>
class Stack {
        T thestack[size];
        int top;
public:
        Stack();
        void push(const T&);
        T pop();
};
```

Example template definition file (*stack.cc*):

```
#include "stack.h"

template <class T, int size> Stack<T, size>::Stack() {
        top=-1;
}

template <class T, int size> void Stack<T, size>::push(const T& item) {
        thestack[++top]=item;
}

template <class T, int size> T Stack<T, size>::pop() {
        return thestack[top--];
}
```

# Template instantiations

## A parameterization of a template is called an *instantiation*.

Here is an example instantiation of template Stack, along with some calls to its member functions:

template instantiation

calls to member functions

```
void stack_test()
{
        Stack<int,20> myintstack;
        int i;

        myintstack.push(5);
        i = myintstack.pop();
}
```

- Each unique instantiation of a template requires code to be generated.
- The compilation system finds the template definition files (for example, *stack.cc*) *based on naming conventions*.
- With most C++ compilation systems, there is no need to name these files explicitly in a makefile nor even build their objects explicitly.

  They are similar to C header files in this respect.

Sample makefile for the example above:

note that there is no mention of *stack.h* nor *stack.cc* in the makefile.

```
main: main.o stack_test.o
    $(C++C) -o $@ main.o stack_test.o
```

# Template instantiations (continued)

## Some compilers instantiate all templates at link time:

Sun SPARCompiler 3.0.1

HP

CenterLine (with ObjectCenter V2)

SGI CC (not DCC/NCC)

Lucid C++

These compilers are based on the *Cfront* front end written at Bell Labs and keep a *repository* of template information (usually in a directory called *ptrepository*).

## Some compilers perform instantiations at compile-time:

(Actually, there is just one compiler in this category, and it perform as many instantiations as possible at compile time but fills in the missing ones at link time.)

Sun SPARCompiler 4.0.1

# Template instantiations (continued)

**Some C++ compilers provide *automatic* instantiation:**

IBM AIX

SGI DCC/NCC

**Some C++ compilers have “manual” template instantiation systems that do not interfere with clearmake:**

Visual C++

Borland C++

OSF?

# Details of the template implementations

## Link-time instantiating systems

- Keep template information in a directory called a *repository* (by default, the name of the directory is *ptrepository*).
- Write to a “type map” file in the repository called *defmap* during each compilation of a file containing a template instantiation (for example, *stack_test.cc*).
- Have a “prelinker” which runs before each link and
  - Reads the *defmap* file to find all the files containing instantiations.
  - Invokes the compiler to instantiate templates when necessary.
  - Writes a set of instantiation files for each template instantiation. For example:

| | |
|---|---|
| `Stack__pt__9_iXCiL220.cc` | *(source code for instantiation)* |
| `Stack__pt__9_iXCiL220.o` | *(object file for instantiation)* |
| `Stack__pt__9_iXCiL220.he` | *(“header cache” file)* |
| `Stack__pt__9_iXCiL220.cs` | *(checksum file)* |

    These files are siblings of the executable produced by the link.

# Details of the template implementations (continued)

## Compile-time instantiating system

- Keeps template information in a repository called (by default) *Templates.DB*.
- At compile time, writes a pair of files for each instantiation. For example:

  | | |
  |---|---|
  | `__0fFStack7i4U_Dpopv.o` | *(object module)* |
  | `__0fFStack7i4U_Dpopv.state` | *(info. about the template instantiation)* |

  These files are siblings of the *.o* file produced by the compilation.
- At link time, gathers the information from the individual *.state* files, writes *Dependency.state* and deletes the individual *.state* files.
- Does something similar for template specializations (details omitted).

# Special makefile macros

**The compilers' template instantiation systems interfere with clearmake's *build avoidance* mechanism.**

- "Sibling derived objects" written to by multiple compilation steps cause extra rebuilds.
- Reuse of template objects from previous builds causes superfluous dependencies to be tracked and interferes with clearmake's wink-in mechanism.
- Some compilers and linkers use timestamp comparisons to determine when to avoid re-instantiation.

**The "special macros" described in Appendix B of the ClearCase Notebook enable C++ systems to work better with clearmake.**

# What happens when you put the macro in a makefile?

The macro expands to an include statement including a makefile from the */usr/atria/config/clearmake* directory:

| Macro | Included makefile |
|---|---|
| `$(CLEARCASE_CMSAFE_CXX_LINK_INST)` | *cmsafe_cxx.mk* |
| `$(CLEARCASE_INCREMENTAL_CXX_LINK_INST)` | *inc_cxx.mk* |
| `$(CLEARCASE_CXX_COMP_INST)` | *comp_cxx.mk* |

- The *.mk* file may also be included directly, but using the macro allows makefiles to be used without changes with clearmake and with other versions of make.
- The macro should be included right at the top of the makefile since it defines other macros which are needed before any invocations of the C++ compiler or linker.
- The macro should appear on a line by itself.
- The definitions in the included makefile have the effect of replacing invocations of the C++ compiler and linker with scripts from the */usr/atria/config/clearmake* directory (for example, *atria_cxx*) which do additional processing before and after invoking the compiler or linker.

# What the special build rules do (common to link-time and compile-time macros)

- Use the *-ptr* compiler option to split the repository into separate directories so that multiple rules won't write to the same files (for example, *ptrepository/defmap*).
- Combine the repositories before the link:
  - The link-time rules use the *atria_make_ptrep* script to combine the type maps before the link.
  - The compile-time rules use multiple *-ptr* options treat one repository as read-write and the rest as read-only.

## Just splitting the repository solves some of problems:

- It splits all incremental files into non-incremental pieces.
- It prevents a derived object from being a sibling of multiple targets so prevents unnecessary rebuilds caused by the compilation systems' writing to files such as *ptrepository/defmap*.

# What the special build rules do for each solution

## Compile-time solution

- Does nothing special beyond splitting the repositories (not much else is needed for this compiler).

## CM safe link-time solution

- Deletes and re-creates the repository each time a program target is built.

The CM safe solution solves these problems beyond the general ones solved by splitting the repository:

- Eliminates dependencies on “old” objects by emptying the repository before each link, which facilitates better sharing (wink-in).
- Eliminates inappropriate timestamp-based build avoidance by the linker, since all template object modules are rebuilt for each link.

Reinstantiating all of the templates for each link may be too inefficient for applications using hundreds of template instantiations, however.

# What the special build rules do for each solution (continued)

## Incremental link-time solution

- Uses the *atria_make_ts_cache* script to a time stamp cache (files with suffix *.ts*).
- Examines whether the template object file and header cache exist and rebuilds if they do not exactly match.

  More details on the exact implementation of this solution may be found in the *atria_make_ts_cache* script itself.

This solution also reduces the inappropriate build avoidance problem.

## How to decide which link-time solution to use

- CM safe is preferable, but may be too inefficient for some users.
- Incremental is faster for a single developer but (at V2.0) prevents sharing.

# How to tell if it's working

## If the macros are working, the output from clearmake should look something like this:

(This example is for the CM safe solution, but all look similar.)

```
======== Rebuilding "hello.o" ========
/usr/atria/config/clearmake/atria_cxx hello.o CC -I/usr/
include -c hello.cc
ATRIA_CXX_CM_SAFE=true
+ export ATRIA_CXX_CM_SAFE
+ /usr/atria/config/clearmake/atria_file_has_suff hello.o
+ /usr/atria/config/clearmake/atria_cxxc hello.o CC -I/usr/
include -c hello.cc
+ /usr/atria/config/clearmake/atria_pre_cxxc hello.o
+ test true = true
+ rm -rf hello.o.ptrep
+ exit 0
+ [ 0 != 0 ]
+ CC -ptrhello.o.ptrep -I/usr/include -c hello.cc
```

## Without the macro, the output would look like this:

```
======== Rebuilding "hello.o" ========
CC -I/usr/include -c hello.cc
```

# Usage models

**We want to understand the common usage models so we can best support them.**

## Questions for discussion:

- Are you currently using C++ with clearmake?
- Are you currently using templates? Do your applications use a lot of templates?
- Which compilers are you using? Are you doing multi-platform development? Development on Windows/Windows NT?
- Are you using the Appendix B solutions?
- Are you building libraries containing templates? Are you exporting the templates or are they contained within the library?
- Do you use a nightly building mechanism?
- Do you rely heavily on wink-ins from clearmake, or do your developers work independently of other work? Does this vary with the type of environment you're using (e.g. UNIX vs. Windows NT)?

## Other questions/issues?