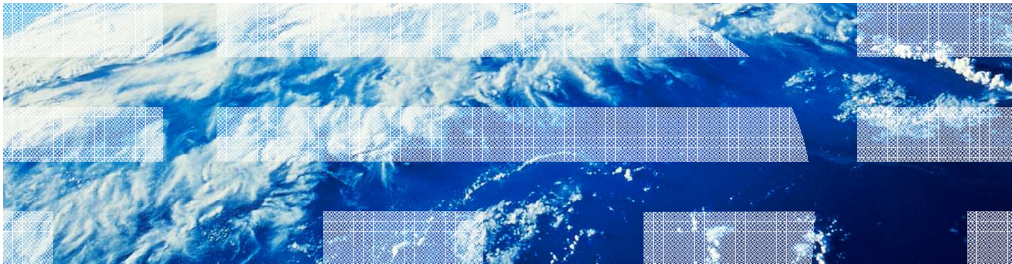
z/OS V1R13

XL C/C++: z/OS V1R13 enhancements



© 2012 IBM Corporation

Session objectives

- Performance Improvement
  - Improved Metal C optimization
  - New hardware built-ins
  - Multiply and Add for hexadecimal types
- Feedback Improvement
  - Informational messages as default in USS
  - Metal C: Function property information
- Usability Improvement
  - Metal C: DSA Support, Argument Parsing
  - C++: Template depth
- Source and Binary Portability Improvement
  - Compatibility Support: Text following #endif, Function attributes (gnu_inline, used, malloc), Temporary lifetimes, Rvalue bindings, Intrinsic complex types, Addressable labels
  - Standards Support (C++0x): Trailing Return Type
- Debugging Support Improvement
  - New Debugging APIs
  - Debugging Inlined Procedures

© 2012 IBM Corporation

Session objectives: Performance improvement

- Improved Metal C optimization
- New hardware built-ins
- Multiply and add for hexadecimal types

© 2012 IBM Corporation

Overview: Improved Metal C Optimization

- Problem Statement / Need Addressed
  - Higher order optimization options were not supported in Metal C.
- Solution
  - Enable the HOT and IPA options for Metal C.
- Benefit / Value
  - The performance of Metal C applications can get a boost with these options.

© 2012 IBM Corporation

Usage and invocation: HOT

- Invoking HOT option:

```
xlc -O3 -qmetal -qhot -S a.c
```

This produces a.s

© 2012 IBM Corporation

IBM

Usage and invocation: IPA

IPA compile phase:

```
xlc -qmetal -qipa -c x.c
```

This produces x.o

```
xlc -qmetal -qipa -c y.c
```

This produces y.o

IPA link phase:

```
xlc -qmetal -qipa -S x.o y.o
```

This produces a.s

Assembly phase:

```
as -mgoff a.s
```

This produces a.o

Bind/Link phase:

```
ld -e //main a.o
```

This produces a.out

© 2012 IBM Corporation

Interactions and dependencies

- None.

© 2012 IBM Corporation

Migration and coexistence considerations

- None.

© 2012 IBM Corporation

IBM

Overview: New hardware built-ins

- **Problem Statement / Need Addressed:**
  - Interlocked-storage-access instructions, available on models where the interlocked-access-facility is installed, provide a means by which a load, update and store operation can be performed with interlocked update in a single instruction. Interlocked storage instructions are not directly available for user source.
- **Solution:**
  - Add support for built-in functions corresponding to the interlocked storage access instructions.
  - Supported interlocked-storage-access instructions are:
    - Load and Add (LAA, LAAG)
    - Load and Add Logical (LAAL, LAALG)
    - Load and And (LAN, LANG)
    - Load and Exclusive Or (LAX, LAXG)
    - Load and Or (LAO, LAOG)
    - Load Pair Disjoint (LPD, LPDG)
- **Benefit / Value:**
  - Source code can now use interlocked-storage-access instructions through built-in functions providing the benefit and speed of the hardware instructions.

© 2012 IBM Corporation

Usage and invocation

- The instructions ending with G operate on 64-bit operands. The built-in functions will follow the same convention. Applications that make use of built-in functions that operate on 64-bit operands will have to be compiled and linked with the LP64 compiler option.

  - int __lad(int* op1, int op3, int* op2); // LAA
  - int __ladg(long* op1, long op3, long* op2); // LAAG
  - int __ladl(unsigned int* op1, unsigned int op3, unsigned int* op2); // LAAL
  - int __ladlg(unsigned long* op1, unsigned long op3, unsigned long* op2); // LAALG
  - int __lan(unsigned int* op1, unsigned int op3, unsigned int* op2); // LAN
  - int __lang(unsigned long* op1, unsigned long op3, unsigned long* op2); // LANG
  - int __lax(unsigned int* op1, unsigned int op3, unsigned int* op2); // LAX
  - int __laxg(unsigned long* op1, unsigned long op3, unsigned long* op2); // LAXG
  - int __lao(unsigned int* op1, unsigned int op3, unsigned int* op2); // LAO
  - int __laog(unsigned long* op1, unsigned long op3, unsigned long* op2); // LAOG
  - int __lpd(unsigned int* op3, unsigned int* op4, unsigned int* op1, unsigned int* op2); //LPD
  - int __lpdg(unsigned long* op3, unsigned long* op4, unsigned long* op1, unsigned long* op2); //LPDG

© 2012 IBM Corporation

Interactions and dependencies

- Hardware Dependencies:

This is an offering for z196 instructions.

- Software Dependencies:

It is implemented under architecture option, ARCH(9).

© 2012 IBM Corporation

Migration and coexistence considerations

- None

© 2012 IBM Corporation

Overview: Multiply and add for hexadecimal types

- Problem Statement / Need Addressed
  - The fused multiply and add instructions are not generated for hexadecimal floating point types.
  - Due to performance and hardware reasons, it is not a good idea to emit these instructions in general cases.
- Solution
  - The new support for zEnterprise makes this feasible to do.
  - Allow FLOAT(MAF) + FLOAT(HEX) for ARCH >= 9.
- Benefit / Value
  - Multiply and Add instructions can be generated potentially increasing performance of the application.

© 2012 IBM Corporation

Usage and invocation

Example: `xlc -qfloat=hex -qfloat=maf -qarch=9 mysource.c`

© 2012 IBM Corporation

Interactions and dependencies

- None

© 2012 IBM Corporation

IBM

Migration and coexistence considerations

- None

© 2012 IBM Corporation

IBM

Session objectives: Feedback improvement

- Informational Messages as Default in USS
- Metal C:
  - Function Property Blocks

© 2012 IBM Corporation

Overview: Informational messages as default in USS

- Problem Statement / Need Addressed
  – When using the INFO option to get diagnostic messages in USS, users have to remember to specify the FLAG(I) option to view the informational severity messages.
- Solution
  – Make FLAG(I) the default in USS as it is in Batch compilation.
- Benefit / Value
  – Users will no longer get the false impression that there is no potential errors in their code if they forget to specify FLAG(I).

© 2012 IBM Corporation

Interactions and dependencies

- None

© 2012 IBM Corporation

IBM

Migration and coexistence considerations

- Any existing compilations that did not turn on FLAG(I) and used the INFO or CHECKOUT options may get new diagnostic messages of informational severity appearing.

© 2012 IBM Corporation

Overview: Metal: Function property blocks (C only)

- Problem Statement / Need Addressed
  - It is hard to find information about functions in Metal generated code.
- Solution
  - Add per-function property data that can be used to identify the C function and the associated properties by code scanning or dump reading.
  - Called Function Property Block (FPB) and can be found via the new Function Entry Point Marker placed immediately before the entry point of each function.
  - Also contains an offset that can be used to find the Prefix Data generated for each compilation unit.
- Benefit / Value
  - Enables Metal C users to find additional information about a function.

© 2012 IBM Corporation

Usage and invocation

- The function property block is generated for all Metal C compiles.
- The function property block is made up of a fixed part (20 bytes in size) followed by a contiguous optional part, with the presence of optional fields indicated by flag bits.
  - Optional fields, if present, are stored immediately following the fixed part of the FPB aligned on fullword boundaries in a given order.
- The detailed layout of FPB is documented in z/OS Metal C Programming Guide and Reference.
- Note: When the COMPRESS compiler option is in effect the function name fields will not be present in the FPB.

© 2012 IBM Corporation

Interactions and dependencies

- None

© 2012 IBM Corporation

Migration and coexistence considerations

- None

© 2012 IBM Corporation

IBM

Session objectives: Usability improvement

- Metal C:
  - DSA Support
  - Argument Parsing
- C++:
  - Template depth

© 2012 IBM Corporation

Overview: Metal: DSA support (C only)

- Problem Statement / Need Addressed
  - A common practice of pointing to the dynamic storage area in user source can be overwritten by compiler generated code.
- Solution
  - A new option DSAUSER is added for requesting a user field of the size of a pointer to be reserved on the stack. This user field can be utilized by the user provided prolog/epilog code for the purpose it was intended for. The user field can be located via the new HLASM global set symbol &CCN_DSAUSER which provides the offset to the user field. The compiler merely allocates the new field on the stack without any code to initialize it.
- Benefit / Value
  - Metal C users now can have a reliable way to get a field of the size of a pointer (i.e. 4 bytes for AMODE 31 and 8 bytes for AMODE 64) on the stack reserved for the user.

© 2012 IBM Corporation

Usage and invocation

Name	DSAUSER | NODSAUSER
Abbreviation	DSAU
Default	NODSAUSER
Category	Object control mode
#pragma option	None
Syntax	`     |- NODSAUSER -|` `>>---- DSAUSER ----------------><`
Description	When DSAUSER is specified with the METAL option, a field of the size of a pointer is reserved on the stack. The user field is a 4-byte field for AMODE 31 and an 8-byte field for AMODE 64. The user field is only allocated if the function has the user supplied prolog/epilog code. The user field can be addressed by using the global set symbol &CCN_DSAUSER ,which is described in z/OS Metal C Programming Guide and Reference. IPA effects: If the DSAUSER option was specified during any of the IPA compile step, it will be applied to all partitions created by the IPA link step.

© 2012 IBM Corporation

New informational global set symbol

Global Set Symbol	Type	Description
&CCN_DSAUSR	Character	The assembly time computed offset to the user field on the stack of the function.

© 2012 IBM Corporation

Interactions and dependencies

- None

© 2012 IBM Corporation

IBM

Migration and coexistence considerations

- None

© 2012 IBM Corporation

Overview: Metal: Argument parsing (C only)

- Problem Statement / Need Addressed
  - The common method of processing command line arguments (argc and argv) in C programs is not natively supported in Metal.
- Solution
  - The “argc” and “argv” format parsing capability is added to Metal C programs. If your main() function uses the standard argc and argv arguments, the Metal C initialization routine is called to parse the raw parameter data received from the hosting environment and to convert the parameter to the standard argc and argv format.
  - If your program is not invoked in the UNIX System Services (USS) environment, you can use the ARGPARSE or NOARGPARSE options to determine if the EXEC PARM needs to be further parsed into individual arguments; the EXEC PARM has to be in this format: a halfword length field followed by a maximum of 100 characters where the length field contains a binary count of the number of bytes in the PARM field. For more information about the ARGPARSE option, see ARGPARSE | NOARGPARSE in z/OS XL C/C++ User’s Guide.
  - If your main() function uses argc and argv arguments and you do not want the parsing to be performed, you can set the new Global Set Symbol &CCN_APARSE to 0 in your prolog code to conditionally bypass the argument parsing.
- Benefit / Value
  - This allows Metal C programs that do need the standard argc and argv style of parameters to have the arguments automatically parsed.

© 2012 IBM Corporation

Usage and invocation

- It is available in all Metal C compiles.
- Requires the need for SCCNOBJ in the bind step.
- User modifiable Global Set Symbols:

Global Set Symbol	Type	Default	Description
&CCN_APARSE	Logical	1	Set to "1" to trigger parser call. Set to "0" to disable parser call.

© 2012 IBM Corporation

IBM

Interactions and dependencies

- None

© 2012 IBM Corporation

IBM

Migration and coexistence considerations

- None

© 2012 IBM Corporation

IBM

Overview: Template depth (C++ only)

- Problem Statement / Need Addressed
  - Immutable limit of 50 recursively instantiated template specializations are processed by the compiler before it halts compilation and emits an error.
- Solution
  - Option TEMPLATEDEPTH with a single integer suboption allows users to specify their own value for how deep they want the compiler to instantiate recursive template specializations.
- Benefit / Value
  - The reason this limit exists is to prevent the compiler from entering infinite loops while instantiating improperly written user template code.
    - This limit has been increased to 300 and now it is controlled by the TEMPLATEDEPTH option.
    - Carefully crafted template code which needs more recursive instantiations is now able to compile.

© 2012 IBM Corporation

Usage and invocation

```
template <int n> void nom() {
  nom<n-1>();
}
template <> void nom<0>() {}
int main() {
  nom<400>();
}
```

- Even a limit on recursive template instantiations of 300 will not be enough to compile this program. However, -qtemplatedepth=400 will now allow this to compile.

© 2012 IBM Corporation

Interactions and dependencies

- None

© 2012 IBM Corporation

Migration and coexistence considerations

- None

© 2012 IBM Corporation

Session objectives: Source and binary portability improvement

- Compatibility Support:
  - Text Following #endif
  - Function Attribute `gnu_inline`
  - Function Attribute `used`
  - Function Attribute `malloc`
  - Temporary Lifetime Extension
  - Binding rvalue to non-const reference
  - Intrinsic Complex Types
  - Addressable Labels
- Standards Support (C++0x):
  - Trailing Return Type

© 2012 IBM Corporation

IBM

Overview: Suppresses diagnostic for text following #else and #endif (1 of 2)

- Problem Statement / Need Addressed
  - C99 standard does not allow extraneous text following #else and #endif.
  - XL C/C++ compiler issues a warning on any extraneous text that is found after #else and #endif.
- Solution
  - A new suboption of LANGLVL, **`textafterendif`**, was implemented to instruct the compiler to suppress the warning for extraneous text following #else and #endif.
- Benefit / Value
  - The warning message is only suppressed if the new suboption of LANGLVL is explicitly specified.
  - The feature enables users to indicate that they want XL C/C++ compiler to be silent about this deviation from the standard, increasing portability.

40

© 2012 IBM Corporation

Overview: Suppresses diagnostic for text following #else and #endif (2 of 2)

Source mysource.c:

```
#ifdef MY_MACRO
#else MY_MACRO not defined
#endif MY_MACRO

int main(void) {
  return 55;
}
```

Compilation command:

```
xlc -qlanglvl=textafterendif mysource.c
```

Returns with no errors.

© 2012 IBM Corporation

IBM

Overview: Function attribute gnu_inline

- Problem Statement / Need Addressed
  - GCC changed the inline keyword behavior to conform to Standard C99 and some existing GCC programs still rely on the old behavior of GCC inline. A new function attribute, **`gnu_inline`**, was introduced by GCC for the user to stick to the old GCC inline behavior.
- Solution
  - Implemented support for the **`gnu_inline`** function attribute to match the GCC inline behavior.
- Benefit / Value
  - The feature makes it easier for the users to port their programs to the XL C/C++ compiler.
  - Note: One key difference in all the various inline behavior is under what conditions a definition of the function is kept and externalized.

© 2012 IBM Corporation

Usage and invocation

Example:

```
extern inline __attribute__((gnu_inline)) nom() {…};
static inline __attribute__((gnu_inline)) bnd() {…};
```


© 2012 IBM Corporation

IBM

Overview: Function attribute used

- Problem Statement / Need Addressed
  - The compiler may remove functions that it does not see used in the program. Ex. The function is referenced only in inline assembly.
- Solution:
  - Function attribute, **`used`**, (as in GCC) is used to instruct the compiler to emit the code for the function even if it appears that the function is not referenced.
  - If attribute **`used`** is used in combination with attribute **`gnu_inline`** (discard the definition), we will replicate GCC's behavior where **`gnu_inline`** wins, and function definition is discarded.
- Benefit / Value
  - Allows keeping functions that would otherwise be removed.
  - Note: For static functions, a definition will be always emitted, if they are marked with attribute **`used`**.

© 2012 IBM Corporation

Usage and invocation

Example:

```
__attribute__((used)) void nom() { }
int main() { nom(); }
```


© 2012 IBM Corporation

Overview: Function attribute malloc

- Problem Statement / Need Addressed
  - Certain functions have properties that can be exploited to increase performance, but there is no way for the compiler to know it. One such property is any non-null pointer returned cannot alias any other pointer that is valid at the time of the function call.
- Solution
  - Function attribute, **`malloc`**, is used to instruct the compiler to treat a function as if any non-NULL pointer it returns cannot alias any other pointer valid when the function returns.
  - The optimization that this attribute enables at the moment will only occur at -O5.
  - Note: This function attribute cannot be used if the user cannot guarantee that the pointer returned by a function points to unique storage. Otherwise the optimization performed may lead to problems at run time.
- Benefit / Value
  - This feature may speed up the execution time of the program since it provides information helpful for extra optimization.

© 2012 IBM Corporation

Usage and invocation

Example:

```
void* nom()  __attribute__ ((__malloc__)) { ... }
```


© 2012 IBM Corporation

Interactions and dependencies

- None

© 2012 IBM Corporation

IBM

Migration and coexistence considerations

- None

© 2012 IBM Corporation

Overview: Temporary lifetime extension (C++ only)

- Problem Statement / Need Addressed
  - A user porting an application from another compiler, which may implement late temporary destruction, wants to extend the lifetime of such temporaries in order to replicate the previous non-standard compliant behavior.
- Solution
  - The compiler option -qlanglvl=tempsaslocals is used to extend the lifetime of C++ temporaries beyond that is specified by the C++ Language Standard in 12.2 [class.temporary].
  - When enabled, the lifetime of temporaries shall be treated as that of local variables declared in the inner-most containing lexical scope where possible.
  - In some contexts temporaries will be treated in the standard compliant way, even when this feature is enabled.
- Benefit / Value
  - When a program incorrectly depends on resources that may have been previously released, this feature might help.

© 2012 IBM Corporation

IBM

Usage and invocation

```
#include<cstdio>

struct S {
  S() { printf("S::S() ctor at 0x%lx.\n", this); }
  S(const S& from) { printf("S::S(const S&) copy ctor at 0x%lx.\n", this); }
  ~S() { printf("S::~S() dtor at 0x%lx.\n", this); }
} s1;

void nom(S s) { }

int main() {
  nom(s1);
  printf("hello world.\n");
  return 0;
}
```

- With -qlanglvl=tempsaslocals, the temporary 's' created for function argument is destroyed after the lexical block of 'main'. By default, 's' is destroyed upon returning from ‘nom'.

© 2012 IBM Corporation

IBM

Interactions and dependencies

- None

© 2012 IBM Corporation

Migration and coexistence considerations

- None

 © 2012 IBM Corporation

Overview: Binding rvalue to non-const reference (C++ only)

- Problem Statement / Need Addressed
  - Non-compliant compilers may allow a non-const reference to be bound to an rvalue.
- Solution
  - Allow a non-const reference to bind to an rvalue only in the declaration of a function parameter or function return type where an initializer is not required and only for user-defined types.
  - This feature also permits an rvalue to bind to a const-volatile reference and it only applies to top-level CV qualifiers on reference types.
- Benefit / Value
  - The option -qlanglvl=compatRValueBinding might accept a Non-compliant program.
  - The option -qinfo=por will enable an informational message indicating that this binding has taken place despite being illegal.

© 2012 IBM Corporation

Usage and invocation

```
struct hey{};
void func(hey& x){}
int main(void)
{
  func(hey());
  return 0;
}
```

- By default, this will be rejected with CCN5295 (S) A parameter of type "hey &" cannot be initialized with an rvalue of type "hey".
- With -qlanglvl=compatRValueBinding, it will compile clean.

© 2012 IBM Corporation

IBM

Interactions and dependencies

- None

© 2012 IBM Corporation

Migration and coexistence considerations

- None

© 2012 IBM Corporation

Overview: Intrinsic complex types (C++ only)

- Problem Statement / Need Addressed
  - Intrinsic complex types were only implemented by C compiler and C++ was lagging this functionality.
- Solution
  - The complex types, float _Complex, double _Complex, and long double _Complex are provided by C++ compiler as built-in types, according to ISO/IEC 9899:1999 Standard.
- Benefit / Value
  - C programs with built-in complex types can now be compiled with C++.
  - Such programs do not need to convert intrinsic complex types to Complex class template implementation in order to compile with C++.

© 2012 IBM Corporation

Usage and invocation

- The feature is normally enabled with -qlanglvl=c99complex.
- The complex types and both unary operators __real__ and __imag__ can be enabled with -qlanglvl=gnu_complex.

```
#include <stdio.h>
#include <complex.h>

int main() {
float _Complex a, b;
a= 2.0f + 3.0f * _Complex_I;
b = 4.0f - 2.0f * _Complex_I;
a = a + b;
printf("a = %f + %f * I . \n", __real__(a), __imag__(a));
}
```

- Compiling with -qlanglvl=gnu_complex produces the following:

a = 6.000000 + 1.000000 * I .

© 2012 IBM Corporation

IBM

Interactions and dependencies

- None

© 2012 IBM Corporation

IBM

Migration and coexistence considerations

- None

© 2012 IBM Corporation

Overview: Addressable labels

- Problem Statement / Need Addressed
  - Source code ported from other platforms or compilers may not work on the z/OS XL C/C++ compiler if it uses addressable labels.
- Solution
  - Add support for the Labels-as-values and Computed-goto features that are implemented by other compilers (Ex. GCC).
- Benefit / Value
  - Makes porting code over to the z/OS XL C/C++ compiler easier.

© 2012 IBM Corporation

Usage and invocation

- mysource.c:

```
int main(void) {
  void* la = &&label1;
  goto *la;
  return 66;
label1:
  return 55;
}
```

- Compile and run:

```
> xlc mysource.c
> ./a.out
```

Returns with code 55.

© 2012 IBM Corporation

IBM

Interactions and dependencies

- None

© 2012 IBM Corporation

IBM

Migration and coexistence considerations

- None

© 2012 IBM Corporation

IBM

Overview: C++0x – Trailing return type

- Problem Statement / Need Addressed
  - Given an expression such as a*b, where a and b are arbitrary types, we cannot say "type of a*b".
- Solution
  - C++0x Trailing Return Type in conjunction with C++0x Decltype removes this limitation.
- Benefit / Value
  - Primary motivation behind Trailing Return Type feature is the ability to declare function templates whose return type depends on the types of the template arguments.

© 2012 IBM Corporation

Usage and invocation

```
template <class A, class B>
decltype(*(A*)(0)**(B*)(0)) multiply (A a, B b)
{
  return a*b;
}
```

- This compiles with -qlanglvl=decltype available in R12, but it introduces code clutter and is error prone.

```
template <class A, class B>
auto multiply(A a, B b)->decltype(a*b)
{
  return a*b;
}
```

- Must more elegant syntax which removes code clutter can now be compiled with -qlanglvl=autotypededuction:decltype or with just -qlanglvl=extended0x.

© 2012 IBM Corporation

Interactions and dependencies

- None

© 2012 IBM Corporation

IBM

Migration and coexistence considerations

- None

© 2012 IBM Corporation

Session objectives: Debugging support improvement

- New Debugging APIs
- Debugging Inlined Procedures

© 2012 IBM Corporation

IBM

Overview: New debugging APIs

- Problem Statement / Need Addressed
  - Debuggers need support to find function entry points.
- Solution
  - CDA provides APIs for DBX to get the entry point of functions as well as the first statement address of each function.
- Benefit / Value
  - The newly added APIs can help the debugger developers to access debug information in the .mdbg and .dbg files more directly.

© 2012 IBM Corporation

Usage and invocation

- ddpi_function_get_func_entrypt

The ddpi_function_get_func_entrypt operation returns the entry point of a function, i.e. its low pc.

```
int ddpi_function_get_func_entrypt(
     Ddpi_Function         function,
     Dwarf_Addr*           ret_func_entrypt,
     Ddpi_Error*           error);
```

- ddpi_function_get_first_stmt_addr

The ddpi_function_get_first_stmt_addr operation returns the address of the first executable statement of a function, i.e. the machine instruction following the function prolog.

```
int ddpi_function_get_first_stmt_addr(
     Ddpi_Function         function,
     Dwarf_Addr*           ret_first_stmt_addr,
     Ddpi_Error*           error);
```


© 2012 IBM Corporation

Interactions and dependencies

- None

© 2012 IBM Corporation

IBM

Migration and coexistence considerations

- None

© 2012 IBM Corporation

IBM

Overview: Debugging inlined procedures

- Problem Statement / Need Addressed
  - Cannot debug inlined procedures. For example, although we can set an entry breakpoint for a procedure, the breakpoint may not be hit if the procedure is inlined. This is because that the debugger can only set breakpoint at the original procedure, not the inline instances.
- Solution
  - We need to provide debug information for each inline instance of a procedure.
- Benefit / Value
  - With the newly generated debug information, the debugger is able to set entry breakpoints at all inline instances so that the user will not miss the breakpoint.

© 2012 IBM Corporation

Usage and invocation

- The inline debug information will be generated with DEBUG(FORMAT(DWARF)) + OPT. For example,

```
xlC -qDEBUG -O2 -o a a.cpp
```

This generates debug information for both the original procedure and the inline instance as follows:

```
<1><  308>      DW_TAG_subprogram
                DW_AT_type                  <146>
                DW_AT_name                  nom
...
<2><  379>      DW_TAG_inlined_subroutine
                DW_AT_abstract_origin       <308>
                DW_AT_low_pc                0x124
```


© 2012 IBM Corporation

Interactions and dependencies

- None

© 2012 IBM Corporation

IBM

Migration and coexistence considerations

- None

© 2012 IBM Corporation

Installation

- Jobs to be run for CBPDO installation:
  - No changes
- PARMLIB statements or members:
  - No changes

© 2012 IBM Corporation

Session summary

- Performance Improvement
  - Improved Metal C optimization
  - New hardware built-ins
  - Multiply and Add for hexadecimal types
- Feedback Improvement
  - Informational messages as default in USS
  - Metal C: Function property information
- Usability Improvement
  - Metal C: DSA Support, Argument Parsing
  - C++: Template depth
- Source and Binary Portability Improvement
  - Compatibility Support: Text following #endif, Function attributes (`gnu_inline`, `used`, `malloc`), Temporary lifetimes, Rvalue bindings, Intrinsic complex types, Addressable labels
  - Standards Support (C++0x): Trailing Return Type
- Debugging Support Improvement
  - New Debugging APIs
  - Debugging Inlined Procedures

© 2012 IBM Corporation

IBM

Appendix - References

- z/OS V1R13 Metal C Programming Guide and Reference (SA23-2225-04)
- z/OS V1R13 XL C/C++ User's Guide (SC09-4767-10)
- z/OS V1R13 XL C/C++ Programming Guide (SC09-4765-12)
- z/OS V1R13 XL C/C++ Language Reference (SC09-4815-11)
- z/OS V1R13 Standard C++ Library Reference (SC09-4949-05)
- z/OS V1R13 Common Debug Architecture User's Guide (SC09-7653-02)
- z/OS V1R13 Common Debug Architecture Library Reference (SC09-7654-04)
- z/OS V1R13 DWARF/ELF Extension Library Reference (SC09-7655-04)
- z/OS V1R13 XL C/C++ Messages (GC09-4819-09)
- z/OS V1R13 XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer (GC09-4913-09)
- z/OS Internet Library: http://www.ibm.com/systems/z/os/zos/bkserv/
- C/C++ Café: http://www.ibm.com/software/rational/cafe/community/ccpp

© 2012 IBM Corporation

Trademarks, disclaimer, and copyright information

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of other IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. IN ADDITION, THIS INFORMATION IS BASED ON IBM'S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM WITHOUT NOTICE. IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION. NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, NOR SHALL HAVE THE EFFECT OF, CREATING ANY WARRANTIES OR REPRESENTATIONS FROM IBM (OR ITS SUPPLIERS OR LICENSORS), OR ALTERING THE TERMS AND CONDITIONS OF ANY AGREEMENT OR LICENSE GOVERNING THE USE OF IBM PRODUCTS OR SOFTWARE.

© Copyright International Business Machines Corporation 2012. All rights reserved.

© 2012 IBM Corporation