What's new in Enterprise PL/I 4.1 and z/OS XL C/C++ V1R12

Peter Elderon (elderon@us.ibm.com)

Chwan-Hang Lee (chwan@ca.ibm.com)

IBM Corporation

February 28, 2011

- IBM zEnterprise 196 (z196) Support
- Enterprise PL/I 4.1 Highlights
- z/OS XL C/C++ V1R12 Highlights
- What's new in Metal C

IBM zEnterprise 196 (z196) Support

ARCH(9) and TUNE(9)

- Both Enterprise PL/I 4.1 and z/OS XL C/C++ V1R12 compilers provide exploitation for z196 with the new ARCH(9) and TUNE(9) options
- This is the result of leveraging the same optimization technology in both compilers.
- ARCH(9) identifies z196 as the target hardware for the program execution and permits exploitation of the new z196 instructions.
- TUNE(9) instructs the compiler to optimize the instruction sequence based on the new z196 microarchitecture.

z196

- The new z196 hardware adds these new facilities to the general instruction set:
  - High-word facility
  - Interlocked-access facility
  - Load/store-on-condition facility
  - Distinct-operands facility
  - Population-count facility
- It is now an Out-Of-Order (OOO) machine.

High-word Facility

- This facility adds a new set of instructions which consider the high-word of the 64-bit GPRs as self-contained 32-bit registers.
- In other words the 16 64-bit GPRs can be viewed as 32 32-bit GPRs.
- Not every 32-bit instruction has a matching one in this facility.
- The high-word cannot be used in address expressions.
- Neither compiler currently exploits this facility.

Interlocked-storage-access Facility

- This facility provides atomic operations such as:

  LOAD AND ADD

  LOAD and bitwise operations

- Neither compiler currently exploits this facility.

Load/store-on-condition facility

- This facility provides instructions to select an operand for load or store based on the condition code.
- This is particularly profitable for conditionally loading a value in (or storing a value from) a register without incurring branch instructions.
- The branchless code sequence has the potential of allowing more optimization opportunities.
- Both compilers currently utilize these instructions.

Load/store-on-condition facility …

- consider this small program:

```
  2.0   | test: proc returns( fixed bin(31) );
  3.0   |
  4.0   |   exec sql include sqlca;
  5.0   |
  6.0   |   dcl c fixed bin(31);
  7.0   |
  8.0   |   exec sql commit;
  9.0   |
 10.0   |   if sqlcode = 0 then
 11.0   |     c = 0;
 12.0   |   else
 13.0   |     c = -1;
 14.0   |
 15.0   |   return( c );
 16.0   | end;
```

Load/store-on-condition facility …

- Under OPT(3) ARCH(8), the instructions after the call are:

```
0000CA  0DEF                 000008 |                BASR     r14,r15
0000CC  5800  D0F4           000010 |                L        r0,<a1:d244:l4>(,r13,244)
0000D0  A718  FFFF           000010 |                LHI      r1,H'-1'
0000D4  EC06  0005  007E     000010 |                CIJNE    r0,H'0',@1L8
0000DA  4110  0000           000010 |                LA       r1,0
0000DE                       000010 |        @1L8    DS       0H
0000DE  58E0  2000           000015 |                L        r14,_addrReturns_Value(,r2,0)
0000E2  5010  E000           000015 |                ST       r1,_shadow1(,r14,0)
```

Load/store-on-condition facility …

- But, under OPT(3) ARCH(9), the instructions after the call are:

```
0000CA  0DEF          000008 |        BASR    r14,r15
0000CC  A718  FFFF    000010 |        LHI     r1,H'-1'
0000D0  BF0F  D0F4    000010 |        ICM     r0,b'1111',<a1:d244:l4>(r13,244)
0000D4  58E0  2000    000015 |        L       r14,_addrReturns_Value(,r2,0)
0000D8  4100  0000    000010 |        LA      r0,0
0000DC  B9F2  8010    000010 |        LOCRE   r1,r0
0000E0  5010  E000    000015 |        ST      r1,_shadow1(,r14,0)
```

Load/store-on-condition facility …

- So, under ARCH(8), the code sequence was:
  - Load SQLCODE into r0
  - Load -1 into r1
  - Compare r0 (SQLCODE) with 0 and branch if NE to @1L8
  - Load 0 into r1
  - @1L8
  - Store r1 into the return value

- While under ARCH(9), the code sequence has no label and no branch:
  - Load -1 into r1
  - Load SQLCODE into r0 via ICM (so that CC is set)
  - Load 0 into r0
  - Load-on-condition r1 with r0 if the CC is zero (i.e. if SQLCODE = 0)
  - Store r1 into the return value

Distinct-operands Facility

- Many traditional instructions operate on two operands and the first operand is replaced with the result. These are referred to as destructive operations.
- This facility introduces a new set of non-destructive instructions where a 3rd operand is added to contain the result.
- This facility includes operations such as ADD, SUBTRACT, SHIFT, AND, OR, etc.
- This allows the compiler more flexibility in register allocation, and therefore the compiler can produce more efficient code.

Distinct-operands Facility

- consider this small program:

```
  2.0       test:
 3.0         proc( a, b, c, d );
 4.0
 5.0         dcl (a,b) fixed bin(31) byvalue;
 6.0         dcl (c,d) fixed bin(31) byaddr;
 7.0
 8.0         c = a - b + a * b;
 9.0         d = a - b - a * b;
10.0
```

Distinct-operands Facility

- Under OPT(3) ARCH(8), these 11 instructions code would be generated:

```
000046  58E0  1008        000008 |        L       r14,_addrC(,r1,8)
00004A  5800  1000        000008 |        L       r0,A(,r1,0)
00004E  1820              000008 |        LR      r2,r0
000050  58F0  100C        000009 |        L       r15,_addrD(,r1,12)
000054  5F20  1004        000008 |        SL      r2,B(,r1,4)
000058  7100  1004        000008 |        MS      r0,B(,r1,4)
00005C  1812              000008 |        LR      r1,r2
00005E  1F20              000009 |        SLR     r2,r0
000060  1E10              000008 |        ALR     r1,r0
000062  5010  E000        000008 |        ST      r1,_shadow1(,r14,0)
000066  5020  F000        000009 |        ST      r2,_shadow1(,r15,0)
```

Distinct-operands Facility

- Under OPT(3) ARCH(9), these 10 instructions code would be generated:

```
000046  58E0  1008          000008 |          L       r14,_addrC(,r1,8)
00004A  5800  1000          000008 |          L       r0,A(,r1,0)
00004E  1820                000008 |          LR      r2,r0
000050  7100  1004          000008 |          MS      r0,B(,r1,4)
000054  58F0  100C          000009 |          L       r15,_addrD(,r1,12)
000058  5F20  1004          000008 |          SL      r2,B(,r1,4)
00005C  B9FA  0012          000008 |          ALRK    r1,r2,r0
000060  5010  E000          000008 |          ST      r1,_shadow1(,r14,0)
000064  B9FB  0002          000009 |          SLRK    r0,r2,r0
000068  5000  F000          000009 |          ST      r0,_shadow1(,r15,0)
```

Distinct-operands Facility

- The key difference is that under ARCH(8), we have:

```
00005C  1812          000008 |          LR      r1,r2
00005E  1F20          000009 |          SLR     r2,r0
000060  1E10          000008 |          ALR     r1,r0
```

While under ARCH(9), the LR is not needed because the generated distinct-operand instruction leaves the “input” operands unchanged and so we have:

```
00005C  B9FA  0012    000008 |          ALRK    r1,r2,r0
000064  B9FB  0002    000009 |          SLRK    r0,r2,r0
```

Population-count Facility

- The new POPCNT instruction provides a count of the number of one bits in each of the eight bytes of the input GPR.
- Each byte in the output GPR contains an 8-bit binary integer in the range of 0-8.
- This can greatly simplify counting the number of 1 bits in a register, and that is useful in cryptography and in other software (such as that counting nucleotide mismatches in DNA strings)
- XL C/C++ provides a new hardware built-in function to support this new instruction, but PL/I 4.1 does not yet

Out-of-Order Microarchitecture

- z196 is the first z/Architecture machine with an Out-of-Order (OOO) design.
- The OOO design allows more instructions on the execution queues to be executed “free”, i.e. the CPU cycle consumed is hidden by the longer-waiting instructions in parallel.
- Under the TUNE(9) option the compiler schedules the instructions to fill the execution queues to maximize OOO and parallelism opportunities.

Performance of C/C++ code on z196

- Programs compiled with the V1R12 compiler may show significant performance improvement when compared to the same programs compiled with V1R11.
- We've seen 11% performance improvement* on a set of CPU intensive integer based C/C++ programs.
- We've also seen 13% performance improvement* on a set of CPU intensive floating-point based C/C++ programs.
- Improvements of 25%* or more were observed in some cases.
- % improvement = (geometric mean of B)/(geometric mean of A), both running on z196, where:
  A = programs compiled by V1R11 targeting the z10
  B = programs compiled by V1R12 targeting the z196

* This is based on internal IBM lab measurements using the following compiler options:

For A: ILP32, XPLINK, HGPR, OPT(3), HOT, IPA(LEVEL(2), PDF, ARCH(8), TUNE(8)

For B: ILP32, XPLINK, HGPR, OPT(3), HOT, IPA(LEVEL(2), PDF, ARCH(9), TUNE(9)

Performance results for specific applications will vary; some factors affecting performance are the source code and the compiler options specified.

Performance of C/C++ code on z196 …

- What if you don't recompile?
- We compared the performance of the same binaries executing on z196 and z10.
  - Binaries were built using the V1R11 compiler.
- On z196 we achieved overall improvements of:
  - 50% for a set of cpu intensive integer based programs*.
  - 125% for a set of cpu intensive floating point based programs*.

* This is based on internal IBM lab measurements using the following compiler options:
ILP32, XPLINK, HGPR, OPT(3), HOT, IPA(LEVEL(2), PDF, ARCH(8), TUNE(8)
Performance results for specific applications will vary; some factors affecting performance are the source code and the compiler options specified.

Performance of C/C++ code on z196 …

- For a detailed description of how to improve your application's performance see also

www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101796

Enterprise PL/I 4.1 Highlights

- Performance enhancement features
- Improved Debug Tool support
- XML validation
- New (sub)options for better quality
- Miscellaneous user requirements

Performance Enhancement Features

REFER

- Code that uses elements of structures with multiple REFERs can be very expensive: each reference uses a costly library call to remap the structure
- Now, for structures where all the elements are byte-aligned, those calls will be avoided and straightforward inline code generated
- If all elements are byte-aligned, no padding is possible and thus the address calculations are relatively simple
- To insure all elements are byte-aligned
  - Specify UNALIGNED on the level-1 part of the declared
  - Declare any NONVARYING BIT as ALIGNED

REFER

- E.g., consider these declares (and note the UNALIGNED):

```
dcl (first,middle,last)    char(*) var;

dcl f_len       fixed bin(31);
dcl m_len       fixed bin(31);
dcl l_len       fixed bin(31);
dcl q           pointer;
dcl
  1 name            based UNALIGNED,
    2 len_first     fixed binary(31),
    2 first         char( f_len refer(len_first) ),
    2 len_middle    fixed binary(31),
    2 middle        char( m_len refer(len_middle) ),
    2 len_last      fixed binary(31),
    2 last          char( l_len refer(len_last) );
```

REFER

- A library call is still made to map the structure for the allocate, but the 6 library calls that would have been done to make the assignments have been eliminated:

```
f_len = length(first);
m_len = length(middle);
l_len = length(last);

allocate name set(q);

q->name.first = first;
q->name.middle = middle;
q->name.last = last;
```

INDEX

- The code generated for the INDEX built-in function has been optimized by Enterprise PL/I when there are only 2 arguments
- The compilers before Enterprise PL/I permitted only 2 arguments, but Enterprise PL/I allows a third argument to specify where the search should start
- This usage has now also been optimized when the second argument is just a single byte, e.g. a semicolon or a blank

INDEX

- This can be very useful in code that processes some text in semicolon delimited chucks, as in:

```
pos = 0;
pos = index( text, ‘;’, pos+1 );
do while( pos > 0 );
  /* process text to semicolon */
  pos = index( text, ‘;’, pos+1 );
end;
```

Improved Debug Tool Support

Reduced object size

- DebugTool uses the statement number table generated by the GONUMBER option (which is why TEST generally forces GONUMBER to be on)
- With Enterprise V3, the GONUMBER table was part of the generated object code (and hence part of the linked load module) even if TEST(SEPARATE) was used
- With Enterprise V4, if you specify TEST(SEP) and GONUMBER(SEP), the compiler will place the statement number table in the side file and thus significantly reduce the size of the generated object
- For compatibility, the default for GN is GN(NOSEP)

Improved automonitor support

- Under Enterprise V3, for its AUTOMONITOR, the compiler generated information that specified only the name of the variable, but omitted any subscripts or pointer qualifications
- Under Enterprise V4, when using TEST(SEPARATE), the compiler generates information that names the fully qualified reference
- E.g., for a statement of the form A(2) = B(2), only A(2) and B(2) will be listed in the monitor window (rather than all of A and all of B).

More support for implicit BASED

- Under Enterprise V3, the compiler generated a symbol table that allowed implicit locator references for variables declared as BASED on simple scalars
- With Enterprise V4, when using TEST(SEPARATE), the compiler will generate information to identify complicated implicit locator references such as those for a variable is BASED on
  - ADDR of array element or
  - other built-in functions (such as ADDRDATA, POINTERADD, etc)

DCL and XREF information

- Under Enterprise V4, when using TEST(SEPARATE), the compiler will include in the side file, information identifying the source lines for
  - declares
  - references (xref refs)
  - Assignments (xref sets)
- This will help enable DebugTool to provide information on these declares and/or to allow you to search for these statements etc

XML Validation

PLISAXD

- The new PLISAXD built-in subroutine is like PLISAXC except that it will cause the incoming XML to be validated
- It requires an additional argument: an Optimized Schema Representation
- Like PLISAXC, PLISAXD uses the System Services XML Parser
- And its arguments are much like PLISAXC:

PLISAXD

- In order, its arguments are
  - An event structure
  - A token passed back to the event functions
  - The address of a buffer containing the XML
  - The size of the buffer
  - The address of the buffer containing the OSR
  - An optional codepage identifier
- The only difference from PLISAXC is the 5th parameter
- The event structure is the same as for PLISAXC

PLISAXD

- While the event structure is the same as for PLISAXC, the exception event may see some additional exceptions found by the validation
- The z/OS Unix command xsdosrg will generate a file containing the OSR for a given schema.
- You must do this before trying to run code using PLISAXD
- And then before invoking PLISAXD, you must read the OSR into a buffer
- The Programming Guide has more details

New (sub)options for Better Quality

DEPRECATE (Racon - MR0427097311)

- The new DEPRECATE option will flag the usage of various include files, built-in functions or variables that you wish to deprecate. It will flag via
  - the BUILTIN suboption, any specified name declared as a BUILTIN
  - the ENTRY suboption, any specified name declared as a level-1 ENTRY
  - the INCLUDE suboption, any specified name used in an %INCLUDE statement
  - the VARIABLE suboption, any specified name declared as level-1 name and not having the BUILTIN or ENTRY attribute

DEPRECATE (Racon - MR0427097311)

- So if you want to flag the usage of UNSPEC and any variable named just I, J, or N, you could specify
  - DEPRECATE(BUILTIN(UNSPEC) VARIABLE(I,J,N))
- Specifying one of the suboptions does not change the setting of any of the other suboptions specified previously. So the above could also be specified as
  - DEPRECATE(BUILTIN(UNSPEC)) DEPRECATE(VARIABLE(I,J,N))

NOGLOBALDO (Telcordia – MR1104096225)

- Under the new RULES(NOGLOBALDO) option, the compiler will flag any DO statement where the loop control variable is declared in a parent procedure – as in this code

```
a: proc;
  dcl jx fixed bin;
  call b;
  b: proc;
     do jx = 17 to 29;
     end;
  end b;
end a;
```

NOGLOBALDO (Telcordia – MR1104096225)

- This usage creates
  - non-transparent code (it is rarely good when a subroutine changes the value of a variable in a parent procedure)
  - less optimized code
- So flagging it is good
- For compatibility, the default is RULES(GLOBALDO)

NOPADDING (Telcordia – MR1110093235)

- Under the new RULES(NOPADDING) option, the compiler will flag any structure where it can tell that there will be padding bytes
- For compatibility, the default is RULES(PADDING)
- RULES(NOPADDING) would flag, for example

```
dcl
  1 a aligned,
    2 b fixed bin(31),
    2 c char(3),
    2 d fixed bin(31);
```

Miscellaneous User Requirements

Init of typed structures (Wuestenrot - MR0312104052)

- In particular, the INIT attribute will now be allowed on leaf elements of a DEFINE STRUCTURE statement
  - However, INIT CALL, INIT TO, and VALUE will still not be allowed on elements of a DEFINE STRUCTURE statement
- For example, the following is now allowed

```
define struct
   1 b,
      2 b1 fixed bin init(17),
      2 b2 fixed bin init(19);
```

Init of typed structures (Wuestenrot - MR0312104052)

- The new VALUE type-function may then be used to initialize or assign to a variable having the corresponding structure type, e.g.

```
define struct
  1 b,
     2 b1 fixed bin init(17),
     2 b2 fixed bin init(19);
define struct
  1 c,
    2 c1 type b init( value(: b :) ),
    2 c2 fixed bin init(23);
dcl x type c static init( value(: c :) );
dcl y type c;  y = value(: c :);
```

Init of typed structures (Wuestenrot - MR0312104052)

- The VALUE function has one mandatory argument that must be the name of a typed structure, and it returns an instance of that typed structure with its initial values
  - If the VALUE function is used with a structure type that is only partially initialized, uninitialized bytes and bits will be zeroed out.
  - The VALUE function may not be used with a structure type containing no elements with the INITIAL attribute

SQL XREF (LVM - MR1112095051)

- The integrated SQL preprocessor will now accept (NO)XREF as an option
- Under XREF, it will produce an XREF listing like that produced by the old SQL precompiler
- This means that the integrated SQL preprocessor provides a full superset of the function available via the SQL precompiler

ONAREA (Telcordia - MR1217095934)

- If AREA has been raised, ONAREA will return a string specifying the AREA reference for which the allocate failed
- So if ALLOCATE X IN(A) fails, ONAREA will return the string ‘‘A”
- And if ALLOCATE X IN(A1.A2(N)) fails, ONAREA will return “A1.A2(N)”

REENTRANT Proc's (StateFarm - MR102909480)

- Before Enterprise PL/I, specifying REENTRANT in the OPTIONS attribute of a PROC statement changed the code that was generated and was required if the code was supposed to be reentrant
- With Enterprise PL/I, it did nothing
- With 4.1, it will now cause the compile to issue a message unless you use either
  - the RENT option, or
  - the DFT(NONASGN) option
- This is under the assumption that such proc's are supposed to be reentrant (and in that case, the compiler should flag any assign to static)

VALUE in structures (MR0213091212)

- The VALUE attribute is now allowed in (non-typed) structures, but then
  - All leaf elements of the structure must have the VALUE attribute
  - The structure must not contain any unions or arrays
- This makes conversion of old declares using STATIC INIT to VALUE easier (and the use of VALUE will let the compiler produce better code)
- It also allows you to have “namespaces” of VALUE

z/OS XL C/C++ V1R12 Highlights

- Source and binary compatibility improvements
- Features for C++0x standard
- Debugging support improvements
- Compiler feedback improvements
- Miscellaneous Enhancements

Source and Binary Compatibility Improvements

typeof keyword

- This gcc keyword is now supported by XL C.
- It was already allowed by XL C++.
- It has the same semantics as the __typeof__ keyword.
- It is invoked by the KEYWORD(typeof) compiler option.
- Or –qkeyword=typeof when using xlc command.

```
int main(void) {
  int rc = 66;
  typeof(rc) returnValue = 55;
  return returnValue;
}
```

New NAMEMANGLING Suboption

- The ANSI name mangling scheme evolves between releases of XL C++ compiler.
- The new NAMEMANGLING(zOSV1R12_ANSI) suboption is added to allow future backward binary compatibility to V1R12 generated binaries for NAMEMANGLING(ANSI).
- The NAMEMANGLING(ANSI) in V1R12 complies with the most recent C++ language features and is equivalent to zOSV1R12_ANSI.
- This new suboption can also be specified using the language directive `#pragma namemangling(…)`

Features for C++0x Standard

C++0x Features

- Variadic Templates
- Delegating Constructors
- Namespace Association
- long long Support
- C99 Preprocessor Changes
- Static Assert
- C99 Compatibility for TR1
- auto
- Decltype
- Extended Friend Declarations

C++0x Features

- These new features are enabled by LANGLVL(extended0x) option.
- The details of each feature can be found in the backup slides section of this presentation.

Debugging Support Improvements

Capture Source Codeset

- The codeset determined by the compiler option LOCALE or ASCII is included in the DWARF sidefile.
- This allows the debugger to display the source code in its original codeset.
- In case demand load is used, the captured source in the .mdbg file will be correct.

Debug parameters in optimized code

- This allows dbx to display the function name and its parameter values when the function is entered.
- This capability is extended to functions produced at higher optimization levels, specifically OPT(2) or OPT(3).
- This is supported only with XPLINK when the STOREARGS suboption is in effect.
- The DEBUG option turns on XPLINK(STOREARGS).

example:

```
(dbx64) where
foo(arg1 = 102, arg2 = 102, arg3 = 102), line 1 in "t.c"
```


SHARE in Anaheim 2011

Compiler Feedback Improvements

Message Severity Modification – C only

- New SEVERITY(I|W|E(msgid)) option allows message level tailoring.
- This allows the customization of severity levels of some of the messages issued by the compiler.
- Informational-level messages can be changed to warning or error to cause non-zero return code from the compile.
- Warning-level messages can be changed to informational or error to tailor the acceptable build conditions.
- Downgrading error messages to lower severity levels does not make sense and is not allowed as the object code produced may have problems if forced through the compile.

Message Severity Modification …

- Example:

```
prototype.c:
int main(void) {
   int * rc = (int*)malloc(sizeof(int));
   foo(rc);
   return *rc;
 }
 int foo(int * rc) {
   *rc = 55;
   return 0;
 }
```

xlc prototype.c -qinfo=pro -qflag=I

INFORMATIONAL CCN3304 ./prototype.c:2 No function prototype given for "malloc".
INFORMATIONAL CCN3304 ./prototype.c:3 No function prototype given for "foo".

xlc prototype.c -qinfo=pro -qflag=i -qseverity=w=CCN3304

WARNING CCN3304 ./prototype.c:2 No function prototype given for "malloc".
WARNING CCN3304 ./prototype.c:3 No function prototype given for "foo".

Improved Aliasing Diagnostics

- Adhering to ANSI aliasing rules allows the compiler to apply safe assumptions during optimization for better performing programs.
- But the ANSI aliasing rule violations can be difficult to find and to understand.
- The new INFO(ALS) option together with the FLAG(I) option trigger the compiler to detect some of the ANSI aliasing rule violations and to issue diagnostic messages.

Improved Aliasing Diagnostics …

- Example:

```
// t.C
int main(int argc) {
  int i = argc;
  short *sp = (short*)&i;
  *sp = 1; // line 4
  return 55;
}
```

xlC t.C -qinfo=als -qflag=i
"./t.C", line 4.3: CCN5590 (I) Dereference may not conform to the current aliasing rules.
"./t.C", line 4.3: CCN5591 (I) The dereferenced expression has type "short". “sp“ may point to “i" which has incompatible type "int".
"./t.C", line 4.3: CCN5592 (I) Check assignment at line 3 column 14 of ./t.C.

Miscellaneous Enhancements

__plo__XXXX Built-in Functions

- These built-in functions provide C/C++ language interface to the sophisticated z/Architecture Perform Locked Operation (PLO) instruction to perform these operations:
  - compare and load
  - compare and swap
  - double compare and swap
  - compare swap and store
  - compare swap and double store
  - compare swap and triple store

__plo__XXXX Built-in Functions …

- There are 24 individual built-in functions with the function name in the format of __plo__XXXX where XXXX corresponds to one of the 24 Function Symbols defined in z/Architecture Principles of Operation.

  for example, __plo__CLG is to generate the PLO instruction for function code 1 – Compare and Load 64-bit operand.

- The ARCHITECTURE(5) option is required to use these functions.
- For operations with 64-bit or 128-bit operands, the LP64 option is required.
- For 128-bit operands, quad-word alignment is required and it has to be managed by the user.

__plo__XXXX Built-in Functions …

- For certain function codes the PLO instruction takes the address to a parameter list which contains all the operands needed to perform the specified operation.
- The layout of the parameter list varies based on function codes.
- To simplify the setup of the parameter list a set of helper macros and typedefs are provided in builtins.h header file.
- For more information please check Chapter 33. “Using hardware built-in functions” in z/OS V1R12 XL C/C++ Programming Guide.

Restrict Parameters – C only

- The restrict keyword was introduced with the C99 standard.
- It allows the programmer to tell the compiler that if the memory addressed by the restrict qualified pointer is modified, no other pointer will access that same memory. This allows the compiler to perform more aggressive optimization.
- Adding the restrict keyword manually to source files can be very time consuming.
- The new RESTRICT compile option provides an easy way to apply the restrict qualifier to all pointer type parameters in the source file being compiled.
- Of course you can only use this if you know all pointer parameters are truly restrict.

Reusable PDF files

- Both stages of Profile-Directed Feedback (PDF) had to be done with identical source file(s) and compiler option(s) or else PDF2 would terminate with an error message.
- The V1R12 compiler relaxes this condition by having PDF2 to tolerate and warn about the out-of-sync conditions from PDF1.
- This reduces the burden of redoing the PDF instrumentation step when only incremental changes are made to an application.
- In particular, this will enable customers doing daily PDF builds to instrument their application on a less frequent basis.

Compiler Infrastructure Improvements

- The optimizer and code generator has gone through reengineering process to facilitate future technology adoption.
- The compilation time may be different for large or complex applications when compiled at higher optimization levels.
- This also applies to Enterprise PL/I 4.

What's New in Metal C

RENT Support

- The RENT option was not available with the METAL option.
- We recognize that a Metal C program deserves the ability to use writable static and external variables while maintaining its reentrancy.
- The V1R12 XL C compiler enables the RENT option for METAL.
- We call it constructed reentrancy for programs with writable static/extern data in that the Writable Static Area (WSA) is dynamically constructed per invocation of the program.
- You have the ability to manage the storage for the WSA when the RENT option is used.
- NOTE 1: The METAL RENT support is independent of and different from NOMETAL RENT support. They should not be mixed.
- NOTE 2: Programs compiled with RENT and NORENT can be mixed as long as the NORENT programs do not call RENT programs.

RENT Support …

- This is the scheme:
  - The static data and extern data are defined in GOFF class M_WSA.
  - The binder builds the WSA image with initialization data from M_WSA definitions found in all object files.
  - The “main” function has a hook after the prolog code to connect to a runtime routine called CCNZINIT.
  - CCNZINIT locates the M_WSA class in the program object and passes the address of M_WSA and its size to a user plug-in routine for storage allocation and initialization.
  - CCNZINIT returns the address of the allocated WSA storage to “main”.
  - “main” saves the returned WSA address .
  - All other functions receives the WSA address in GPR 0.
  - On exit from “main” CCNZTERM is called for cleanup.

RENT Support …

- IBM supplies default plug-in routines for WSA storage management.
- The default plug-in routine (CCANWSAI) issues this macro for both AMODE 31 and AMODE 64:
  `STORAGE OBTAIN,LENGTH=(n),BNDRY=PAGE`
- Unless you want to allocate the storage in other ways, the default should be sufficient.
- The AMODE of the runtime routine is the same as the AMODE of “main”.
- Likewise, the user plug-in is assumed to be the same AMODE as the runtime routine.
- If your program has mixed AMODEs, you need to ensure the WSA storage is addressable to all AMODE 31 functions.

RENT Support …

- For AMODE 31, the runtime routines are called CCNZINIT and CCNZTERM.
- For AMODE 64, the runtime routines are called CCNZQINIT and CCNZQTRM.
- The default plug-in routines are called CCNZWSAI and CCNZWSAT for AMODE 31.
- And they are called CCNZQWSI and CCNZQWST for AMODE 64.
- The object code for these routines are supplied in the CBC.SCCNOBJ dataset.
- Thus it is necessary to add this dataset to the binder SYSLIB allocation when linking your Metal C RENT program.

RENT Support …

- The runtime routines assume that NAB is supplied by function “main”, i.e. they use the same stack storage as “main”.
- This provides the opportunity, for example if the stack storage was obtained from CICS by “main”, the Metal RENT runtime routines will also operate on CICS storage.
- Allocating 1K of extra space for NAB should be sufficient for CCNZINIT and CCNZTERM as well as CCNZWSAI and CCNZWSAT. For AMODE 64, consider 2K.

RENT Support …

- Interface to the WSA initialization plug-in routine:

```
typedef void * (init_func_t)(void * wsa_image_addr, unsigned long
wsa_size, void **user_info_addr, unsigned int alignment);
```

- Input parameters:

  wsa_image_addr - address of the WSA image in the program object
  wsa_size - total size of the application's WSA
  user_info_addr - address to a pointer field for saving your own pointer
  alignment - the minimum required alignment of the allocated WSA storage. For example, alignment=8 means double-word alignment.

- Return value:

  The address of the allocated and initialized WSA storage.

RENT Support …

- Interface to the WSA termination plug-in routine:

```
typedef void (term_func_t)(void * allocated_wsa_addr, unsigned
long wsa_size, void * user_info_addr);
```

- Input parameters:

  allocated_wsa_addr - address of the allocated WSA storage
  wsa_size - total size of the application's WSA
  user_info_addr - the saved user pointer

RENT Support …

- New Global Set Symbols:

  `&CCN_MAIN` – to identify if the function is “main”
  `&CCN_RENT` – to identify if compiled with the RENT option

  `&CCN_WSA_INIT` – to provide your WSA initialization routine name
  `&CCN_WSA_TERM` – to provide your WSA termination routine name

  For example:

```
          GBLC &CCN_WSA_INIT
          GBLC &CCN_WSA_TERM
&CCN_WSA_INIT SETC 'MYWSAI'
&CCN_WSA_TERM SETC 'MYWSAT'
```

RENT Support …

- With the RENT support, it is now possible to use Metal C, as an alternative to assembler, to write programs to run in CICS environment.
- You can use Metal C to write CICS applications using CICS API.
- You can also use Metal C to write CICS exit routines using CICS exit programming interface (XPI).
- There are sample programs documented in the “Metal C Programming Guide and Reference” to show a CICS version of the “hello, world” program and an exit program.

Learn more at:

- IBM Rational software
- IBM Rational Software Delivery Platform
- Process and portfolio management
- Change and release management
- Quality management
- Architecture management
- Rational trial downloads
- developerWorks Rational
- IBM Rational TV
- IBM Rational Business Partners
- IBM Rational C/C++ Cafe

© **Copyright IBM Corporation 2010. All rights reserved.** The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials 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 the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. Product release dates and/or capabilities referenced in these materials may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way. IBM, the IBM logo, the on-demand business logo, Rational, the Rational logo, and other IBM products and services are trademarks of the International Business Machines Corporation, in the United States, other countries or both. Other company, product, or service names may be trademarks or service marks of others.

Backup Slides

C++0x Feature Details

Variadic Templates [3]

- It is for generic containers with any number of parameters.
- Type and non-type template parameters can now be specified as type and non-type parameter packs.
- Template parameter packs can be instantiated with 0 or more arguments.
- In a function template, the template parameter pack can be used to create a function parameter pack.
- Generic containers do not need to be designed with a hard-coded number of template parameters.
- This allows easier implementation of generic containers such as tuples or templates to represent function objects.

Variadic Templates …

- Example 1 (template class):

```
// template parameter pack <class… A>
template <class...A> struct container{};

container<> a1;
container<int> a2;
container<int,char,float> a3;

template <bool...B> struct container1{};

container1<> b1;
container1<true> b2;
container1<true,false,true,true> b3;
```

Variadic Templates …

- Example 2 (template function):

```
#include <cassert>
template <class A, class B> struct container{};

// arg is a function parameter pack
template <class... C, class... D> int func( container<C,D>... arg ){
   // sizeof… returns the size of the function parameter pack
   assert( sizeof...(arg) == sizeof...(container<C,D>));
   return sizeof...(container<C,D>);
}

struct a1{}; struct a2{};

int main(void){
   container<a1,a2> a;
   assert( func( a,a,a,a,a ) == 5);
   return 0;
}
```

Variadic Templates …

- Example 3 (pack expansion and access to pack members):

```
#include <iostream>
using namespace std;

template <class X> struct test{
   test(X data){ cout << "ctor test: " << data <<endl;}
};

template <class... A> void func(A... arg1){
  test<int> data(99);}

template <class head, class... tail>
void func(head arg1, tail... arg2){
   test<head> local(arg1);   // arg1 is the first argument
   func(arg2...);   // arg2… is a pack expansion containing
                    // all the remaining args
}
```

Variadic Templates …

```
int main(void){
  func();
  // match the 1st version of func
  func();
  // match the 2nd version of func
  func(1);
  // match the 2nd version of func
  func(1,2,3,4);
  return 0;
}
```

Output:

ctor test: 99
ctor test: 1
ctor test: 99
ctor test: 1
ctor test: 2
ctor test: 3
ctor test: 4
ctor test: 99

Delegating Constructors [4]

- This allows Common initializations in multiple constructors of the same class to be concentrated in one place in a robust, maintainable manner.
- This should make the program more readable and maintainable.
- This should also reduce the code size.

Delegating Constructors …

- Example 1 (simple positive execution):

```
#include <cstdio>
template <typename T, typename U> struct A {
   const T t;
   const U u;
   static T tdef;
   static U udef;
   A(T t_, U u_) : t(t_ ^ u_), u(u_) { }
   A(T t_) : A(t_, udef) { }
   A(U u_) : A(tdef, u_) { }
};
template <typename T, typename U> T A<T, U>::tdef;
template <typename T, typename U> U A<T, U>::udef;
```

Delegating Constructors …

```
int main(void) {
  A<unsigned char, unsigned>::tdef
   = 42u & 0x0F;
  A<unsigned char, unsigned>
   a(42u & 0xF0);
  std::printf("%d\n", a.t);
  return 0;
}
```

Output:

42

Delegating Constructors …

- Example 2 (delegating constructor with more than one initialiser):

```
struct A {
   int x, y;
   A() : y(10), A(42) { }
   A(int x) : x(x), y(0) { }
};
```

Compiler Diagnostics

"./t.C", line 3.10: CCN8439 (S) The constructor initializer is unexpected. This constructor delegates at line 3, column 16.

"./t.C", line 3.16: CCN8441 (I) "A::A()" delegates to "A::A(int)".

This can be fixed by moving initialization of y from initialization list to the constructor body of delegating constructor.

Delegating Constructors …

- Example 3 (delegating constructor that delegates to itself):

```
struct A {
   int x, y;
   A() : A(42) { }
   A(int x_) : A() { x = x_; }
};
```

Compiler Diagnostics

"./t.C", line 4.4: CCN8440 (S) "A::A(int)" delegates to itself.

"./t.C", line 4.4: CCN8441 (I) "A::A(int)" delegates to "A::A()".

"./t.C", line 3.4: CCN8441 (I) "A::A()" delegates to "A::A(int)".

This can be fixed by removing the entire initialization list from the second delegating constructor on line 4.

SHARE in Anaheim 2011

Namespace Association [2]

- This is to allow namespace association with inline namespace definitions.
- Members of an inline namespace to be used as if they were members of another namespace.
- This gives library vendors the ability to use the same source and object files (including interface headers and library archives) for all of the implementations.

Namespace Association …

- Example (Use in library versioning with explicit specialization):

```
foo.h
namespace SomeLibrary {

#ifdef SOME_LIBRARY_USE_VERSION_2_
   inline namespace version_2 { }
#else
   inline namespace version_1 { }
#endif

   namespace version_1 {
      template <typename T>
      int foo(T a) { return 1; }
   }
   namespace version_2 {
      template <typename T>
      int foo(T a) { return 2; }
   }
}
```

```
#include <foo.h>
#include <iostream>

// Client code
struct MyIntWrapper {   int x;};

// specialize SomeLibrary::foo() using
// the correct version
namespace SomeLibrary {
   template <> int foo(MyIntWrapper a)
{ return a.x; }
}

int main(void) {
   using namespace SomeLibrary;
   MyIntWrapper intWrap = { 4 };
   std::cout << foo(intWrap) +
                foo(1.0) << std::endl;
}
```

long long Support [7]

- C++0x Standard formally introduced the ‘long long’ type for integers that are larger than what can be represented in 4 bytes.
- The XL C++ compiler already had the support for ‘long long’ type.
- The ‘long long’ type is now included in the EXTENDED0X language level.
- The new C++0x integral promotion rules produce a different result from the existing IBM ‘long long’ type promotion rules.
- A diagnostic message is issued to alert the user when the use of ‘long long’ differs from the C++0x standard.

long long Support …

- **Example** (difference in promotion rules)

```
int main(void)  {
   // LONG_MAX is 2147483647
   long long x =  2147483648;
}
```

Due to promotion rules differences, this code produces the following Informational message:

"./t.C", line 3.19: CCN8928 (I) Integral constant "2147483648" has implied type unsigned long int under the non-C++0x language levels. It has implied type long long int under C++0x.

C99 Preprocessor Changes [8]

- The C++0x standard “imported” some C99 preprocessor rules that were not in C++98.
- These changes make C++ and C99 more compatible with each other.
- The _Pragma operator.
- Increased maximum limit for #line preprocessor directive.
- New predefined macros.
- Wide and narrow string concatenation.

C99 Preprocessor Changes …

- Example 1 (_Pragma and #line):

  The following statements are 100% functionally equivalent.

```
#pragma comment(copyright, "IBM 2007")
_Pragma("comment(copyright, \"IBM 2007\")")
```

  The #line directive previously could not accept a value greater than 32767, but this limit has been increased to 2147483647.

C99 Preprocessor Changes …

- Example 2 (new predefined macros):

  `__STDC__`

  macro defined to 0 for all language levels (i.e. It is ALWAYS defined to 0)

  `__STDC_HOSTED__`

  macro is defined to 1 for LANGLVL=EXTENDED0x, and undefined otherwise

C99 Preprocessor Changes …

- Example 3 (wide and narrow string concatenation):
  - Prior to C++0x, you could not concatenate two strings together if they were not both either “wide” or “narrow”.
  - The resulting strings will default to “wide” but they can be casted back to narrow.

```
int main() {
  char* narrow  = (char*) "This string " L"will be narrow";
  wchar_t* wide =         "This string " L"will be wide";
}
```

Without compiling this program under C++0x, both these string assignments are in error.

Static Assert [6]

- This is a facility to enforce template parameter constraint.
- An assert macro tests assertions at runtime.
- An #error preprocessor directive is processed before templates are instantiated.
- This should improve support for library building by allowing libraries to detect common usage errors at compile time.

Static Assert …

- Example (assertion on template parameter type):

```
template <typename T> void foo(T s) {
  static_assert(sizeof(s) == 2, "foo not instantiated with short");
}
int main() {
  short s = 2;
  foo(2);
  return 0;
}
```

"./t.C", line 2.9: CCN7520 (S) "foo not instantiated with short"
"./t.C", line 1.28: CCN5700 (I) The previous message was produced while processing "foo<int>(int)".
"./t.C", line 6.9: CCN5700 (I) The previous message was produced while processing "main()".

Static Assert …

- No binary compatibility issues with existing programs.
- Keyword static_assert was NOT reserved by C++0x Standard so users could have source code using this keyword as identifier.
- In extended0x language mode XL C++ will diagnose this keyword if it is not used as static assert.
- Static assert can also be enabled by default which is handy for library writers (enabled without -qlanglvl=extended0x).

```
__extension__ static_assert(0,"user message 1");
__static_assert (0,"user message 1");
```

C99 compatibility for TR1 [5]

- This supplies the missing C99 compatibility layer for TR1.
- The following C/C++ headers are now exporting C99-specific symbols (macros, functions and function overloads): `<complex.h>`, `<math.h>`, `<cctype>`, `<cmath>`, `<cstdio>`, `<cstdlib>`, `<cwchar>`.
- The following new C++ headers have been added: `<cfenv>`, `<cinttypes>`, `<cstdbool>`, `<cstdint>`.
- The exported symbols are placed in the namespace `std::tr1`.
- TR1 support is in closer agreement with the library described in C99.

C99 compatibility for TR1 …

- Example 1 (type-generic function template):

```
#include <cmath>
int main()
{
    double dbl = 0.0;
    long lng   = 1L;
    std::tr1::signbit(dbl);  // OK
    std::tr1::signbit(lng);  // Errror
}
```

C99 macro ‘signbit’ is implemented as function template in TR1 with template parameter constraint of accepting floating point types only.

C99 compatibility for TR1 …

- Example 2 (overloads):

```
#include <math.h>

float flt                 = 1.0F;
double dbl                = 2.0;
long double ldbl          = 3.0L;
long long llng            = 4LL;
long lng                  = 5L;
int i                     = 6;
short shrt                = 7;
unsigned long long ullng  = 8ULL;
unsigned int ui           = 9U;
Unsigned short ushrt      = 10U;
```

```
pow(flt, dbl);   // returns double
pow(flt, i);     // returns double
pow(i, i);       // returns double
pow(i, lng);     // returns double
pow(ushrt, shrt);// returns double
pow(flt, ldbl);  // returns long double
pow(llng, ldbl); // returns long double
pow(shrt, ldbl); // returns long double

sin(flt);        // returns float
sin(ullng);      // returns double
sin(ui);         // returns double
sin(ldbl);       // returns long double
```

Note that all function calls in **bold** received “no best match” error prior to this feature.

auto

- Redesigned auto keyword to let the compiler automatically deduce the variable's type by looking at its initialiser.
- Eliminate need to specify type explicitly whenever an initialiser is used.
- Very useful with multiple layers of templates.

auto …

- Example 1:

```
for (typename vector<T>::const_iterator iter = v.begin();
  iter!=v.end(); ++iter) {...}
```

can be replaced with:

```
for (auto iter = v.begin(); iter!=v.end(); ++iter) {...}
```

The auto type notation makes the code much more elegant to write and it removes redundancy of specifying type when the compiler is able to deduce the type.

auto …

- Example 2 (template argument):

```
template<class T, class U>
void foo(const vector<T>& vt, const vector<U>& vu)
{ auto tmp = vt[i]*vu[i];}
```

  - It is difficult to write code without using auto type in cases where the type of the variable depends on template argument.
  - The type of variable tmp is what you get from multiplying **T** by **U**, but exactly what that is can be hard for the human reader to figure out.
  - Compiler is able to deduce type of variable tmp once it handles particular **T** and **U.**

auto …

- Meaning of C ++ keyword **auto** introduces source compatibility issue.
  - auto type when auto type deduction is turned on
  - auto storage class specifier in non-C++0x mode

```
int main() {
// non-C++0x language level: OK, auto represents storage
// C++0x language level: ERROR, auto represents type
  auto int r;
}
```

Decltype [10]

- This keyword allows declarations to be defined by the type of an arbitrary expression.
- Primary designed for automatically type deduced declarations for function return types but useful in general.
- Can be used with conjunction with C++0x auto keyword.

Decltype …

- Example:

```
foo<int>::someType someVar1;
decltype(someVar1) someVar2; // someVar2 is the same type as
  someVar1

template <typename T, typename _T>
decltype( (*(T*)0) * (*(_T*)0) ) foo (const _T& arg1, const T&
  arg2)
{ return arg1 * arg2; }
// return type deduced by multiply operator

template <typename T, typename _T>
void performGenericWork(T t, _T _t) { auto f = foo(t, _t); }
// auto declaration deduced by function call
```

Extended friend declarations [11]

- The class keyword may be omitted on friend declaration.
- This allows generic programming: “friend T”, where 'T' is a template parameter, which may be a class, typedef, or primitive type.
- The 'friend T' does not inject a new declaration in the way 'friend class T' does.
- The extended friend replaces legacy behaviour of oldfriend support with similar friend syntax but different semantics.

Extended friend declarations …

- Example:

```
class C;
typedef C Ct;

class X1 {
friend C;       // C++98 Warns about missing 'class' keyword
};

class X2 {
friend Ct;      // C++98 emits an Error
friend D;       // Error under both C++98 and C++0x
friend class D;
};

template <typename T> class R {
friend T;       // C++98 emits an Error
};

R<C> rc;
R<int> Ri;
```

Links to C++0x draft papers

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2857.pdf
[2] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2535.htm
[3] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2242.pdf
[4] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1986.pdf
[5] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf
[6] http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2004/n1720.html
[7] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1811.pdf
[8] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1653.htm
[9] http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2002/n1385.htm
[10] http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1478.pdf
[11] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1791.pdf