New C/C++ and PL/I releases

Visda Vokhshoori
visdav@ca.ibm.com

z/OS V2.2 XL C/C++

z/OS V2.2 XL C/C++ availability

- Preview announce date, January 2015
- Availability announce date, July 2015
- General availability date, September 30, 2015

Source: If applicable, describe source origin

z/OS V2.2 XL C/C++ highlights

- Support for the new hardware
- Asm and hardware model support
- Two new high performance libraries
- Improved functionality

Support for the new hardware

ARCH(11)

- To enable the compiler with the z13 feature we are introducing new sub options for ARCHITECTURE and TUNE options
- ARCH(11) will prompt the compiler to utilize the list of hardware instructions new in z13
- TUNE(11) will prompt the compiler to improve run time performance according to the micro-architecture characteristics of z13

Support for Load Halfword Immediate on Condition

- The new instructions LOCHI and LOCGHI are like LHI and LGHI except that the load depends on the conditional code and the mask in the instruction
- The immediate value, as indicated in the name of the instruction, is halfword in size, so this applies only to values that fit in 16 bits
- The compiler will exploit this under OPTimize and ARCH(11)

Support for Convert from|to Packed

- The new instructions
  - Convert from Packed
    - CDPT – Convert to long DFP
    - CXPT – Convert to extended DFP
  - Convert from Packed

    CPDT – Convert from long DFP

    CPXT – Convert from extended DFP

Under OPT and ARCH(11) the back-end utilizes these instructions where it finds the right pattern; these will improve the run-time performance as the DFP instructions are performed in registers and in general faster than packed decimal

Support for the Vector Facility

- Vector programming support provides programmers direct access to SIMD instructions from the z13 Vector Facility for z/Architecture
- Language extensions based on the AltiVec Programming Interface specification with suitable changes and extensions

Support for the Vector Facility

- A new VECTOR option
- macro: __VEC__
- Compile options: -qARCH(11) -qVECTOR
- FLOAT(AFP(NOVOLATILE)) and TARGET(zOSV2R2) are assumed default
- Vector data types:
  - {vector, vector} {bool, signed, unsigned} {char, short, int, long long}
  - {vector, vector} double
- Various language extensions on the vector types for natural usage just like for the native types
  - Ex.
    - Assignment operator (=)
    - Address operator (&)
    - Pointer arithmetic
    - Unary operators (++, –, +, -, ~)
    - Binary operators (+, -, *, /, %, &, |, ^, <<, >>)
    - Relational operators (==, !=, <, >, <=, >=)

Support for the Vector Facility

- Comprehensive set of vector built-in functions for access and manipulation of individual vector elements
  - –In the following high-level categories:
    - Arithmetic
    - Compare
    - Compare ranges
    - Find any element
    - Gather and scatter
    - Generate Mask
    - Isolate Zero
    - Load and Store
    - Logical
    - Merge
    - Pack and unpack
    - Replicate
    - Rotate and shift
    - Rounding and conversion
    - Test
    - All Predicates
    - Any Predicates

Support for the Vector Facility

- Example:

```
#include <builtins.h>
#include <stdio.h> int main() {
  vector signed int a = {-1, 2, -3, 4};  // declare and initialize a vector with 4 signed integer elements
  vector signed int b = {-5, 6, -7, 8};
  vector signed int c, d;     // declare vectors with 4 signed integer elements


  c = a + b;              // Generates VAF
  d = vec_abs(c);      // Generates VLPF

  printf("d[0] = %d\n",d[0]); // prints 6 -- d[0] extract the 1st element from the vector
  printf("d[1] = %d\n",d[1]); // prints 8
  printf("d[2] = %d\n",d[2]); // prints 10 printf("d[3] = %d\n",d[3]); // prints 12

  return 0;
}
```

Support for the Vector Facility

- AutoSIMD compiler optimization
- Identifies source statements that are safe and profitable to be transformed into vector form, i.e. using single instruction that operates on multiple data

- Example:
- unsigned int i,n,x;
- unsigned int *a, *b;
- for (i=0; i<n; ++i) {
- a[i] = a[i] + 4*b[i];
- }

- AutoSIMD can process the loop 4x faster.

Support for the Vector Facility

- AutoSIMD compiler optimization is turned on by default when
- HOT,FLOAT(AFP(NOVOLATILE)),ARCH(11)

- This optimization makes use of the z13 vector facility. Using the instructions that can operate on multiple data streams, introduced in z13 compiler can produce parallel code to improve throughput; more data processed in the same window of time

- Workloads that can benefit from SIMD include string processing-intensive workloads, security and cryptographic workloads, and mathematical modeling workloads

0x000000c2	466.389252	MVHI 2192(GPR4),X'7'	MVHI 2160(GPR4),X'7'	513.183472	0x000000bc
0x000000c8	20.575991	MVHI 2196(GPR4),X'7'	MVHI 2164(GPR4),X'7'	27.369781	0x000000c2
			MVHI 2168(GPR4),X'7'		0x000000c8
			MVHI 2172(GPR4),X'7'	437.916565	0x000000ce
			MVHI 2176(GPR4),X'7'	27.369781	0x000000d4
			MVHI 2180(GPR4),X'7'	20.527336	0x000000da
			MVHI 2184(GPR4),X'7'	513.183472	0x000000e0
			MVHI 2188(GPR4),X'7'	41.054680	0x000000e6
			MVHI 2192(GPR4),X'7'	1019.524536	0x000000ec
			MVHI 2196(GPR4),X'7'	6.842445	0x000000f2
0x000000ce	13.717325	MVHI 2200(GPR4),X'7'	MVHI 2200(GPR4),X'7'	123.164040	0x000000f8
0x000000d4	425.237274	XC 2256(88,GPR4),2256(GPR4)	XC 2256(88,GPR4),2256(GPR4)	855.305847	0x000000fe
0x000000da	13.717325	VREPIF VR0,7			
0x000000e0	500.682587	VREPIF VR2,2			
0x000000e6	6.858659	VST VR0,2160(,GPR4)			
0x000000ec	89.162651	VST VR2,2208(,GPR4)			
0x000000f2	1063.093140	VST VR0,2176(,GPR4)			
0x000000f8		L GPR6,2396(,GPR4)	L GPR6,2396(,GPR4)	34.212227	0x00000104
0x000000fc	41.151989	LARL GPR7,*+6780	LARL GPR7,*+6728	957.942505	0x00000108
0x00000102	541.834595	VST VR2,2224(,GPR4)			

Initialization of 40 bytes starting at GPR4+2160 with 7 has been SIMDized, Reduced path length, more data processed

Support for the Vector Facility

- Vector programming references:
  - SIMD Business Analytics Acceleration on z Systems

[http://www.redbooks.ibm.com/abstracts/redp5145.html?Open]

  - z/OS V2.2 XL C/C++ Programming Guide

[http://publibz.boulder.ibm.com/epubs/pdf/cbc1p210.pdf
Chapter 35: Using vector programming support]

  - AltiVec Technology Programming Interface Manual

[www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf]

asm and hardware model support

Ability to insert asm statements

- Clients have asked us to support HLASM statements for all C or C++ programs and not just with Metal C
- Now you can inline hardware instructions in your LE enabled C or C++ program using the __asm keyword
- Compiler output is program object, doesn't generate assembly source like MetalC
- Requires certain level of HLASM PTF

Ability to insert asm statements

- New options and sub-options

  –**ASM**: causes __asm, __asm__to be statements

  –**Keyword(ASM)**: gives asm the same semantic as __asm

  –**ASMLIB**: specifies the macro libraries to be used when assembling the inline assembler source code; libraries will be concatenated

  Example: -qASMLIB=A -qASMLIB=B

  //ASMLIB DD DISP=SHR,DSN=A

  // DD DISP=SHR,DSN=B

- New messages will be reported by the compiler under message CCN1148

Ability to insert asm statements

```
//jobname JOB acctno,name...
//COMPILE EXEC PGM=CCNDRVR,
// PARM='/SEARCH(''CEE.SCEEH.+'') NOOPT SO OBJ ASM KEYWORD(ASM) ASMLIB(//SYS1.MACLIB)'
//STEPLIB DD DSNAME=CEE.SCEERUN,DISP=SHR
// DD DSNAME=CEE.SCEERUN2,DISP=SHR
// DD DSNAME=CBC.SCCNCMP,DISP=SHR
// DD DSNAME=SYS1.SASMMOD1,DISP=SHR
//SYSLIN DD DSNAME=MYID.MYPROG.OBJ(MEMBER),DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSIN DD DATA,DLM=@@
#include <stdio.h>
   ...
int my_cipher(char* in, char* out, int* len, int parmBlock, int* mode) {
/* Inline CIPHER Message */
  asm("REDO KMC %3,%0\n"
      " BRC 3,REDO\n":"+XL:RP:e"(x),"+r"(*len),"+XL:NR:r1"(parmBlock), \
       "+XL:RP:e"(out):"XL:NR:r0"(*mode):);
}
@@
//SYSUT1 DD DSN=...
   ...
//*
```

Support for inline asm

- z/OS V2.2 XL C/C++ User's Guide
  - New options ASM and ASMLIB

[Chapter 4 http://publibz.boulder.ibm.com/epubs/pdf/cbc1u210.pdf]

z/OS V2.2 XL C/C++ Language Reference

– [Chapter 7 http://publibz.boulder.ibm.com/epubs/pdf/cbc1l210.pdf]

Mixing ARCH levels in one compile

- Before, you had to put code using different ARCH levels into separate functions – and you had to pay for the expense of calling the appropriate function
- You can now have code using different ARCH levels within one compilation unit

Mixing ARCH levels in one compile

- Now this code can be inlined with the new
  - **#pragma arch_section(<architecture>)**
- The pragma indicates the start of a section of the source intended for the machine indicated by <architecture>
- The compiler switches to architecture specified, and at the end the section switches back to the previous architecture

Identifying the hardware model and features

- Programs may need to check the machine model before doing some processing
- Three new builtins help with this:
- builtin_cpu_init(void)
  - –Runs the CPU detection code, and saves the CPU information in a compiler defined/managed buffer
  - –Must be called at least once before either of the following is invoked

Identifying the hardware model and features

- builtin_cpu_is(const char* cpumodel)

  –Returns 1 if the CPU is of type *cpumodel (“5”, “6”, …, “11”)*

- builtin_cpu_supports(const char* feature)

  –Returns 1 if the CPU supports one of the features indicated

  –*feature* values are

  –`"longdisplacement", "etf2", "etf3", "dfp", "prefetch", "storeclockfast", "loadstoreoncond", "popcount", "interlocked", "tx", "dfpzoned", "vector128", "5", ..., "11"`

Identifying the hardware model and features

xlc -c –qARCH=9 –o popcnt.o popcnt.c
xlc -c –qARCH=5 –o main.o main.c
xlc –o a.out popcnt.o main.o
./a.out

>cat main.c

```
// Counts bits that have the value of 1 in a byte
unsigned long myBytePopcount(char op)
{
    unsigned long count = 0;
    for (int i = 0; i < 8; i++) {

        if ((op & 1) == 1) {
            count++;
        }
        op >>= 1;
    }
    return count;
}
```

Identifying the hardware model and features

//main.c continued

```
#define arch9 2217
int main() {
struct utsname ruon;
  uname(&runon);
int archLevel = atoi(runon.machine);
  unsigned long output;
  unsigned long input = 55;
```

```
  //Check architecture level
  if (archLevel >= arch9) {
     output = callArch9Builtin(input);
  } else {
    output = myBytePopcount(input);
  }
  printf("%lu\n", output);
  return 0;
 }
```

Identifying the hardware model and features

>cat popcnt.c

```
#include <builtins.h>
//Compiled with ARCH(9)
unsigned long callArch9Builtin(unsigned long op) {
  return __popcnt(op);
}
```

Using the #pragma arch_section, and run-time check builtins the above can be simplified to one file.

>xlc –qARCH=5 –o a.out main.c

```
#include <builtins.h>
unsigned long myPopCountOnByte(unsigned long op)
{
     unsigned int count = 0;
     for (int i = 0; i < 8; i++) {

          if ((op & 1) == 1) {
               count++;
          }
          op >>= 1;
     }
     return count;
}

int main()
{
     unsigned char input = 55;
     unsigned long output;
     __builtin_cpu_init();
     if (__builtin_cpu_supports("popcount")) {
          #pragma arch_section(9)
          {
               output = __popcnt(input)&0xFF;
          }
     } else {
          output = myPopCountOnByte(input);
     }
     printf("%lu\n", output);
     return 0;
}
```

Identifying the hardware model and features

- z/OS V2.2 Language Reference

[Chapter 18 #pragma arch_section
http://publibz.boulder.ibm.com/epubs/pdf/cbc1l210.pdf]

- z/OS V2.2 Programming Guide

[Chapter 34 __builtin_cpu_xxxx
http://publibz.boulder.ibm.com/epubs/pdf/cbc1p210.pdf]

New high performance libraries

Insert Custom Session QR if Desired.

New high performance math library

- MASS (Mathematical Acceleration Sub-System) library
- A comprehensive set of elementary/special mathematical functions (e.g. exp, log, sin, etc.) tuned for high performance on zEC12 and z13
- 3 kinds of libraries:
  - 59 scalar functions
    - Easiest to use in existing code since names match existing runtime library functions
  - 77 vector functions
    - Generally provides the highest performance, provided vector_length is sufficient (approximately >2 to >10 depending on the function)
  - 8 SIMD (z13 only) functions
    - Convenient for code written to use vector datatypes and built-in functions
- Single- and Double-precision FP, in IEEE

New high performance math library

MASS archives /usr/lpp/cbclib/lib:

```
libmass.arch10.a        libmassv.arch10.a        libmass.arch11.a
libmassv.arch11.a       libmass_simd.arch11.a
```

MASS header files /usr/include:

```
mass.h        mass_simd.h        massv.h
```

Mass in batch mode:
MASS archives:

CBC.SCCNM10	scalar MASS library overlap with LE
CBC.SCCNN10	scalar and vector MASS library not overlap with LE
CBC.SCCNM11	single precision MASS ARCH(11) library
CBC.SCCNN11	double precision MASS ARCH(11) library

MASS Header files:
CEE.SCEEH.H

Compile Options:
FLOAT(IEEE)
ARCHITECTURE(10) -the minimum for zEC12/zBC12
ARCHITECTURE(11) -the minimum for the z13
VECTOR -if using the mass_simd
NOEXH -if using C++
Do not change the rounding mode from the default value: ROUND(N)
Include the appropriate header files
If using scala MASS, include both mass.h and math.h
If using vector MASS, include massv.h
If using SIMD MASS, include mass_simd.h

MASS performance

- z/OS MASS (Mathematical Acceleration Subsystem) vector functions on z13 demonstrate **up to 2.9x higher throughput** than the corresponding z/OS V2.1 XL C/C++ runtime library math functions on zEC12.

- A key subset of MASS vector functions are heavily used in Analytics and are accelerated using SIMD instructions.These functions demonstrate **up to 6.8x higher throughput** than the corresponding z/OS V2.1 XL C/C++ runtime library math functions on zEC12.

Disclaimer
This claim is based on results from internal lab measurements. A subset of the MASS vector functions is accelerated using SIMD instructions on z13. The SIMD benefit is demonstrated using this subset. The performance improvements achieved will vary depending on the workload and other factors.

Source: If applicable, describe source origin

New high performance linear algebra library

- ATLAS (Automatically Tuned Linear Algebra Software) library
- A high-performance versions of all the BLAS (basic linear algebra subprograms) routines, and a subset of the LAPACK (linear algebra package) routines
- Tuned for high performance on zEC12/zBC12 and z13
- Both single and multi-threaded versions available, with IEEE floating point
- Supplied libraries
  - ATLAS main libraries
    - ATLAS specific variants of the BLAS, CBLAS, and LAPACK routines
  - CBLAS libraries
    - C interface versions of the BLAS routines
  - LAPACK libraries
    - C interface versions of the LAPACK routines
  - Fortran BLAS libraries
    - Fortran 77 interface versions of the BLAS routines
  - Supports 31-bit C linkage, 31-bit XPLINK, and 64-bit XPLINK

New high performance linear algebra library

- ATLAS is provided on USS only
- Library & header file location:
  - Library: /usr/lpp/cbclib/lib/atlas/
  - Header: /usr/lpp/cbclib/include/atlas/

- Compile options requirement
  - FLOAT(IEEE)
  - ROUND(N) -default rounding for FLOAT(IEEE)
  - ARCHITECTURE(10) -the minimum required ARCH level
  - ARCHITECTURE(11) -required to enable vector functionality
  - VECTOR -required to enable vector functionality

- More information can be found:
  - z/OS V2.2 Programming Guide

[Chapter 43 http://publibz.boulder.ibm.com/epubs/pdf/cbc1p210.pdf]

  - External ATLAS web site

[http://math-atlas.sourceforge.net]

ATLAS performance

- A key subset of z/OS ATLAS 3.10.0 (Automatically Tuned Linear Algebra Software) double precision functions on z13 demonstrate **up to 44% higher throughput** than the corresponding functions on zEC12.

- Selected key z/OS ATLAS 3.10.0 (Automatically Tuned Linear Algebra Software) functions are accelerated using SIMD instructions. These functions demonstrate **up to 80% higher throughput** on z13 than the corresponding functions on zEC12.

Disclaimer:
This claim is based on results from internal lab measurements. The double precision function improvement is derived from comparisons of a select set of commonly used z/OS ATLAS 3.10.0 functions executing on z13 to the equivalent functions executing on zEC12. A subset of these functions is accelerated using SIMD instructions on z13. The SIMD benefit is demonstrated using this subset. The performance improvements achieved will vary depending on the workload and other factors.

Source: If applicable, describe source origin

Increased functionality

Insert Custom Session QR if Desired.

Improved make

- 'make' dependency file generation through -M did not include non-existent headers or allow named targets
- New make options address these and similar concerns:

- -MT allows setting the dependent target name

- -MQ is -MT but also escapes 'make' special characters for easier dependency file usage

- -MG allows missing header files to be included in the dependency list

- The -qmakedep=pponly suboption will run the include preprocessing only
  - A dependency file will be generated
  - But no object code will be generated

Expanded user-reserved space on the Metal C stack

- This is an enhancement to an existing feature
- The option DSAUSER, available only with MetalC, allows the user to reserve a pointer on the stack
- However, the user may want to leave more space on the stack

- DSAUSER has been enhanced to allow you to specify how much space to reserve on the stack

  –xlc -qDSAUSER=12

  - This will reserve a space equal to the size of 12 words on the stack

Eliminating a null pointer check

- The C++11 ANSI Standard requires
  - –checking the pointer returned from the placement new operator
  - –and performing the initialization only when the pointer is not null
- The check for null pointer returned from other operators new and new[] is not required but performed
- New sub-option `LANGLVL(NOCHECKPLACEMENTNEW)` will remove the check of the pointer returned by the placement new operator
  - –And can thus speed up the program by eliminating unnecessary null pointer checks

Improvements to dbgld utility

- A new option has been added to prompt dbgld to capture source files without executable statement, e.g. variable declarations
  - –dbgld -cf a.out
  - –JCL CAPSRC(FULL)

Improvements to c89 utility

- Enable c89 to pass environment variables to the binder
- Accept mixed case, longer than 9, and hyphenated symbol or entry point name to be specified to SYMTRACE, and EP binder options

```
int one_TWO_three() { return 1+2+3; }
export IEWBIND_OPTIONS="SYMTRACE=one_TWO_three"
c89 -Wl,'MSGLEVEL=0' t.c >o 2>&1
grep one_TWO_three o
 IEW2420I A61B SYMTRACE: SYMBOL one_TWO_three IS DEFINED IN SECTION
  $PRIV000010
 IEW2422I A61D SYMTRACE: SYMBOL one_TWO_three DEFINITION ORIGINALLY CAME FROM
```

Enterprise PL/I 4.5

Insert Custom Session QR if Desired.

Enterprise PL/I 4.5 highlights

- Improved performance
- Enhanced middleware support
- Increased string length limit
- Introduced support for JSON
- Increased functionality
- Added features to enforce code quality
- Satisfied 28 RFE's

Improved performance

Insert Custom Session QR if Desired.

ARCH(11)

- Under ARCH(11), the compiler will exploit
  - The new Load Halfword Immediate on Condition instruction
  - The new vector hardware instructions (and registers)
- The latter significantly improves the performance of the code generated for (MEM)SEARCH and (MEM)VERIFY of CHAR and WIDECHAR strings
- The vector facility consists of:
  - Support instructions
  - Integer instructions
  - String instructions
  - Float instructions

ARCH(11)

- PL/I currently exploits only the support and string instructions
- In particular, to implement SEARCH and VERIFY, these the vector instructions are very useful
  - find_any_element_equal
  - find_any_element_not_equal
  - string_range_compare

ARCH(11)

- For example, this simple code that tests if a UTF-16 string is numeric

```
wnumb: proc( s );

   dcl s  wchar(*) var;
   dcl n  wchar value( '0123456789' );
   dcl sx fixed bin(31);

   sx = verify( s, n ); if sx > 0 then
   ...
```

- Is done with an expensive library call with ARCH <= 10

ARCH(11)

- With ARCH(11), the vector instruction facility is used to inline it as

```
E700   E000   0006                VL       v0,+CONSTANT_AREA(,r14,0)
E740   E010   0006                VL       v4,+CONSTANT_AREA(,r14,16)
                          @1L2    DS       0H
A74E   0010                       CHI      r4,H'16'
4150   0010                       LA       r5,16
B9F2   4054                       LOCRL    r5,r4
B9FA   F0E2                       ALRK     r14,r2,r15
E725   E000   0037                VLL      v2,r5,_shadow1(r14,0)
E722   0180   408A                VSTRC    v2,v2,v0,v4,b'0001',b'1000'
E7E2   0001   2021                VLGV     r14,v2,1,2
EC5E   000D   2076                CRJH     r5,r14,@1L3
A74A   FFF0                       AHI      r4,H'-16'
```

Other performance improvements

- Much faster code is now generated for MOD and REM of large FIXED DEC
- Previously, calls to a library routine were used for this
- Now inline code using DFP makes the calculation much faster
- ARCH(11) is not required for this

Other performance improvements

- ARCH(11) not required is also not required for improvements to

- INLIST of CHAR(1)
- BETWEEN for CHAR(1) and WCHAR(1)
- SEARCH and VERIFY of WCHAR(1)

Other performance improvements

- A SELECT statement of the form

```
select( x ); when( '..' ) ..
when( '..' ) ..
...
```

- Was turned into a (fast) branch table if x was CHAR(1) and into a (slow) series of string compares if x had length > 1

- Now it will also be turned into a branch table if x is CHAR(2) or CHAR(4)

Other performance improvements

- One user has some code with this SELECT statement

```
SELECT(PLAUS.PLZ); WHEN('0000') ...
WHEN('9999') ...
WHEN('1000') ...
WHEN('1004') ...
```

- With more than 2500 WHEN clauses

- With 4.4, it takes 90 seconds to compile under OPT(2)
- With 4.5, it takes 5 seconds, and the generated code is much better, too!

EXEC CICS statements

- The code generated for every EXEC CICS statement consists of a call to a CICS entry point with a first parameter that is an unprintable character string often longer than 100 bytes
- This string encodes for CICS what the statement is requesting
- Since the call appears to be an ordinary call to the compiler, it allocates a temporary on the stack and generates a MVC instruction to copy the 100+ bytes from the constant area to that temporary

EXEC CICS statements

- The 4.5 compiler now marks the first parameter of the CICS entry point with the INONLY attribute
- This eliminates the allocation of the temporary on the stack and the (slow) MVC instruction to copy to it
- This means the code will run faster, and your DSA will be smaller too

Enhanced middleware support

Structures as indicator variables

- Previously, if you wanted to use indicator variables with a structure in an EXEC SQL statement, you had to name each element of the structure and an associated indicator variable. For example, given

```
dcl 1 h3, 2 a fixed dec, 2 b fixed dec, ..., 2 z fixed dec;
```

- You had to code a matching structure and then name everything

```
dcl 1 in3, 2 a fixed bin(15), 2 b fixed bin(15), ..., 2 z fixed dec; exec

sql insert into mytable

        values( :h3.a:in3.a, :h3.b:in3.b, ..., :h3.z:in3.z );
```

Structures as indicator variables

- Not fun and not easy to maintain or enhance. But now you can use a structure as indicator variable. So, given
- you can use the matching indicator structure and name only the structures

```
dcl 1 h3, 2 a fixed dec, 2 b fixed dec, ..., 2 z fixed dec;

dcl 1 in3, 2 a fixed bin(15), 2 b fixed bin(15), ..., 2 z fixed dec; exec

sql insert into mytable

        values( :h3:in3 );
```

Structures as indicator variables

- But it gets better: with the new INDFOR attribute (it's like LIKE except the copied names all get the FIXED BIN(15) attribute), the matching indicator structure is easy to declare. So, given

```
dcl 1 h3, 2 a fixed dec, 2 b fixed dec, ..., 2 z fixed dec;
```

- you can use INDFOR and code simply

```
dcl 1 in3 indfor h3;
exec sql insert into mytable

         values( :h3:in3 );
```

Structures as indicator variables

- And multi-row fetch (and dimacross) work here, too! So, given

```
dcl 1 h3(3) dimacross, 2 a fixed dec, 2 b fixed dec, ..., 2 z fixed dec;
```

- you can use INDFOR and code the very simple

```
dcl 1 in3(3) dimacross indfor h3; exec sql

insert into mytable
        values( :h3:in3 );
```

Named constants as host variables

- You can now use named constants as SQL host variables if
  - DB2 allows a simple, unamed constant at that place in the EXEC SQL statement
- And
  - the named constant has either the attribute
    - CHARACTER, in which case its VALUE attribute must specify a character string
  - or
    - FIXED, in which case its VALUE attribute must specify a decimal number or an expression that can be reduced to an integer constant

Statement validation

- Previously, when the SQL preprocessor scanned an EXEC SQL statement, it would report only the first error in the statement
- Now, it will report all the errors in every EXEC SQL statement

SQL CODEPAGE option

- Using the new SQL preprocessor option (NO)CODEPAGE, you can control the preprocessor's use of the compiler's CODEPAGE option when it sets the CCSID of a host character variable
- When CODEPAGE is in effect, the compiler's CODEPAGE option is always used as the CCSID for SQL host variables of character type
- When NOCODEPAGE is in effect, the compiler's CODEPAGE option is used as the CCSID for SQL host variables of character type only if the SQL preprocessor option NOCCSID0 is also in effect
- NOCODEPAGE is the default for compatibility with previous releases

SQL WARNDECP option

- Using the new SQL preprocessor option (NO)WARNDECP, you can control whether the SQL preprocessor issue a warning message if it uses the DB2-provided DSNHDECP module
- PP(SQL('WARNDECP')) matches what the previous releases did
- But NOWARNDECP is now the default (since this message is almost meaningless to most users and hence is just distracting noise)

Less noise from the SQL preprocessor

- Message DSNH4760I is now suppressed – this shows up as

IBM3024I I …… DSNH4760I DSNHPSRV The DB2 SQL Coprocessor is using the level 2 interface under DB2 V9

- This is almost always uninteresting
- This has been removed for 4.3 as well as 4.4

Increased string length limit

From 32K to 128M

- Previously no string could be 32K or longer, and VARYING strings had only a 2-byte length prefix
- The new VARYING4 attribute will let you declare strings as having a 4-byte length prefix
- The new STRING suboption of the LIMITS option will let you set the threshold for string lengths to be 32K, 512K, 8M, or 128M

From 32K to 128M

- The maximum length is one less than the threshold
- The default is 32K
- The threshold values may be specified using K or M suffices or as decimal numbers, i.e. as 32K or 32768 or as 128M or 134217727
- But ...

From 32K to 128M

- A threshold bigger than 32K may be specified only if the CMPAT(V3) option is also in effect – because an expanded string descriptor is needed
- And mixtures of code compiled with CMPAT(V2) and CMPAT(V3) face the same restrictions as mixtures of code compiled with CMPAT(V1) and CMPAT(V2) faced 30-years ago
- Fwiw, the new VARYING4 attribute may be used with CMPAT(V2) - but then the length must still be less than 32K

Long string considerations

- Specifying LIMITS(STRING(n)) where n > 32K may also greatly increase the amount of stack storage used by your code
- For example, in PUT LIST(A || B) where A and B are CHAR(*) VARYING4, the compiler will allocate a temporary on the stack equal to the maximum string size – so under LIMITS(STRING(128M)) this would be 128M off the stack!!
- The MAXTEMP compiler option will alert you to such statements

Long string considerations

- The STRING limit applies to all kinds of strings: BIT, CHAR, GRAPHIC and WIDECHAR
- The VARYING4 attribute is also supported for all kinds of strings
- ALIGNED VARYING strings are halfword-aligned – ALIGNED VARYING4 strings will be fullword-aligned

CMPAT(V3) considerations

- The descriptors generated under CMPAT(V3) are different than those generated under CMPAT(V2) – and not just for strings
- The offsets in structure descriptors and the bounds etc in array descriptors are all 8-byte integers

Support for JSON

JSON overview

- A series of built-in functions provide the ability to
  - Generate JSON text
  - Parse JSON text
  - Validate JSON text

JSON generation

- All JSON written out will be in UTF-8 with the compiler and library handling any necessary conversions from EBCDIC
- a series of "put" functions are provided and all have a buffer address and buffer length as their first 2 arguments, and all return the number of bytes written
- attempts to write variables containing data types incompatible with JSON will be flagged at compile time
- escaped characters will be created as needed

JSON parsing

- All JSON to be parsed must be in UTF-8 with the compiler and library handling any necessary conversions to EBCDIC
- a series of “get” functions are provided and all have a buffer address and buffer length as their first 2 arguments, and all return the number of bytes read
- attempts to read variables containing data types incompatible with JSON will be flagged at compile time
- whitespace characters will be skipped over when found

JSON

- For example, suppose we have this sample JSON text

```
{ "passes“ : 3, "data“ :
  [

          { "name“ : "Mather",  "elevation“ : 12100 }
        , { "name“ : "Pinchot", "elevation“ : 12130 }
        , { "name“ : "Glenn",   "elevation“ : 11940 }
    ]
}
```

- And that it is in a buffer at address p and of length n

JSON

- If we had a corresponding PL/I structure

```
dcl
   1 info
      2 passes fixed bin(31), 2 data(3),
         3 name char(20) varying,
         3 elevation fixed bin(31);
```

- Then *jsonGetValue(p, n, info)* will by itself fill in the whole structure
- This one simple function call would do all the work for you

JSON

- But if we did not know how many data instances we would get, our PL/I structure might instead look like

```
dcl
   1 info based(q)
     2 count fixed bin(31),
     2 data( passes refer(count) ),
       3 name char(20) varying,
       3 elevation fixed bin(31);
```

- And it would have to be dynamically allocated – but this is still easy:

JSON

- Four built-in references would suffice:

```
rd = jsonGetObjectStart(p,n);          /* read over {                          */
rd = jsonGetMember(p+rd,n-rd,passes);  /* read “passes”:3 and assign it        */
allocate info;
rd = jsonGetComma(p+rd,n-rd);          /* read over ,                          */
rd = jsonGetValue(p+rd,n-rd,info.data); /* read “data” … and assign it         */
```

- And this works, of course, no matter how much whitespace is present

Increased functionality

Insert Custom Session QR if Desired.

INLIST

- This built-in function is useful in determining if a value belongs to a set of values and allows you to put a SELECT in the middle of an IF
- It requires a minimum of 3 arguments and accepts a maximum of 64
- INLIST(x, a, b, c, …) is equivalent to (x = a) | (x = b) | (x = c) …
- All the arguments must have computational type
- The compiler will optimize this when possible

INLIST

- The arguments can be numbers, strings, or arbitrary expressions
  - if inlist(stadt, ‘Berlin’, ‘Bern’, ‘Rom’, ‘Wien’) then
  - if inlist(uppercase(stadt), ‘BERLIN’, ‘BERN’, ‘ROM’, ‘WIEN’) then
  - if inlist(x, 2, y, 7, z) then
  - if inlist(b, b1 | b2, ‘1100’b, b3 & b4) then
- These would all be converted to a short-circuited series of compares

INLIST

- But if the first argument is “nice” and the rest are all similar values, then the compiler will turn the inlist reference into a branch table. For example,
  - inlist(x, 2, 3, 5, 7, 11, 13, 17, 19)
- would become a branch table if x is FIXED BIN(p,0) with p <= 31 or if X is FIXED DEC(p,0) with p <= 9
- The values 2, 3, 5, etc don’t have to be literals – they can be named constants (VALUEs) or restricted expressions
- And if all are CHAR(1), a simple table look-up is generated

INLIST

- Branch tables would also be built if
  - X is BIT(n) with 1 <= n <= 16 and the other arguments are bit constants
  - X is CHAR(1) and the other arguments are CHAR(1) constants
- Again the second and subsequent arguments don't have to be literals – they can be named constants (VALUEs) or restricted expressions

BETWEEN

- This built-in function is useful in determining if a value is in an interval
- It requires exactly 3 arguments
- BETWEEN(x, a, b) is equivalent to (x >= a) & (x <= b)
- All the arguments must be ordinals or have real numeric type
- The compiler will optimize this when possible
  - For example, if x, a, and b are all FIXED BIN(p,0) with p <= 31, then the compiler will turn BETWEEN(x, a, b) into one comparison (not two!)
  - OORDINAL, CHAR(1), and WCHAR(1) are optimized in the same way

NULLENTRY

- This built-in function allows you to initialize an entry variable with a null value – including static variables

- Dcl function_pointer limited entry static init(nullentry());

- You can also use it to test an entry value to see if it is null

PLISTCK, PLISTCKE, and PLISTCKF

- These built-in subroutiness will generate the corresponding instructions

- PLISTCK(x) sets an UNSIGNED FIXED BIN(64)
- PLISTCKE(x) sets a CHAR(16) NONVARYING
- PLISTCKF(x) sets an UNSIGNED FIXED BIN(64)

- Each returns a FIXED BIN(31) that is the condition code set by the corresponding hardware instruction

SMFTOJULIAN and JULIANTOSMF

- These built-in functions convert between the Julian and SMF date formats
- SMFTOJULIAN(d) converts a CHAR(4) SMF date to a CHAR(7) YYYDDDD
- JULIANTOSMF(d) converts a CHAR(7) YYYDDDD to a CHAR(4) SMF
- No error checking is done at run-time for these functions – the conversions are done in-line, and the input data must be valid

PLISAX

- These built-in functions now allow the event function pointers to be null
- When null, the event will not be called
- This allows you to write smaller, faster XML parsing code
- It requires a library PTF (but not the 4.5 compiler)

REINIT

- This statement allows you to reset a variable with its INITIAL values
- The variable must be level-one, unsubscripted with storage class
  - AUTOMATIC
  - BASED
  - CONTROLLED
  - STATIC

SYSDIMSIZE

- This preprocessor built-in function returns the number of bytes needed to hold the largest array bound

- Under CMPAT(V1), SYSDIMSIZE returns 2
- Under CMPAT(V2), SYSDIMSIZE returns 4
- Under CMPAT(V3), SYSDIMSIZE returns 8

SYSPOINTERSIZE and SYSOFFSETSIZE

- These preprocessor built-in functions return the size (in bytes) of a POINTER and an OFFSET
- As of now, they always return a 4
- But when 64-bit code is supported they could return an 8, and they will be useful in writing code that will compile and run correctly both in 32-bit and in 64-bit mode

SYSPOINTERSIZE and SYSOFFSETSIZE

- For example, you could use syspointersize to declare C's malloc

```
%if syspointersize = 8 %then %do;
  define alias size_t  fixed bin(63);
%end; %else %do;
  define alias size_t  fixed bin(31);
%end;

dcl malloc  ext('malloc')
            entry( type size_t byvalue )
            returns( byvalue pointer )
            options( linkage(optlink) nodescriptor );
```

- And this would be correct for 32- and 64-bit

XML compiler option

- This option now supports an XMLATTR suboption with suboptions of APOSTROPHE or QUOTE
- It determines whether XML attributes are enclosed as '…' or “....”
- It requires a library PTF (as well as the 4.5 compiler)

Quotes in pictures

- The apostrophe symbol is now accepted in PICTURE specifications
- With the same usage as the comma or period symbol
- This requires only a library PTF

Added features to enforce code quality

Insert Custom Session QR if Desired.

RULES(NOLAXRETURN)

- Previously RULES(NOLAXRETURN) caused the compiler to generate code to raise the ERROR condition if
  - RETURN; was hit in a PROC with the RETURNS attribute
  - RETURN(...); was hit it a PROC without the RETURNS attribute
- Now the options has been enhanced so that the compiler will raise the ERROR condition if code falls through to the END statement in a PROC with the RETURNS attribute

RULES(NOLAXRETURN)

- This makes it much easier to detect the lack of the desired statement after the else in this segment of real customer code

```
func: proc( . . . ) returns( fixed bin(31) );

  . . .

  if flags = ''b then; else
    return(0);
end;
```

- This problem would probably now be resolved within minutes rather than days it took to resolve it previously via the PMR process

MAXBRANCH option

- The new MAXBRANCH compiler option lets you find blocks (PROCEDUREs and BEGIN-blocks) that are perhaps too complex. More precisely, it flags any block that has too many conditional branches
- A statement of the form "if a then ...; else ..." adds 1 to the total number of branches in its containing block, and a statement of the form "if a = 0 | b = 0 then ..." adds 2.
- SELECT statements and conditional DO loops also to the total
- The default is MAXBRANCH(2000)

NONASSIGNABLE option

- The new NONASSIGNABLE suboption of the DEFAULT compiler option now supports INONLY and STATIC as suboptions
- NONASSIGNABLE(INONLY) specifies that parameters declared with the INONLY attribute are given the NONASSIGNABLE attribute.
- NONASSIGNABLE(STATIC) specifies that STATIC variables are given the NONASSIGNABLE attribute.
- These suboptions have no effect on variables that are explicitly given the ASSIGNABLE or NONASSIGNABLE attribute or on structure members that have inherited the ASSIGNABLE or NONASSIGNABLE attribute from a parent where it was explicitly specified.

NONASSIGNABLE option

- BYVALUE parameters are given the INONLY attribute after the resolution of the (NON)ASSIGNABLE attribute, and hence the NONASSIGNABLE(INONLY) suboption has no effect on BYVALUE parameters (unless, of course, they are explicitly given the INONLY attribute).

- To specify that both STATIC and INONLY variables are to be given the NONASSIGNABLE attribute, then you must specify the suboption NONASSIGNABLE(STATIC INONLY).

- The NONASSIGNABLE attribute may be specified without any suboptions, in which case it has the same meaning as in previous releases, namely NONASSIGNABLE(STATIC).

RULES(NOLAXQUAL)

- The NOLAXQUAL suboption of RULES now accepts another choice of suboptions:
- Under the default NOLAXQUAL(ALL), the compiler will flag all violations of the qualifications rules (either STRICT or LOOSE)
- But under NOLAXQUAL(FORCE), the compiler will flag only those violations when the element belongs to a structure with the new FORCE(NOLAXQUAL) attribute
- This gives you the ability to enforce mandatory qualification on a structure-by-structure basis

RULES(NOLAXNESTED and NOPADDING)

- The new ALL | SOURCE suboptions to the RULES(NOLAXNESTED) and RULES(NOPADDING) compiler options provide finer control over when the compiler flags questionable coding
- Under ALL, all violations are flagged
- Under SOURCE, only those violations that occur in the primary source file are flagged
- This matches what is already supported for RULES(NOUNREF)
- For each of these options, ALL is the default