Enterprise COBOL V6.3 was announced! What's New?

Tom 'Captain COBOL' Ross
Feb 24, 2020
Session 26536

Agenda

- Announce/GA key dates & offerings
- Major features
  - z15 exploitation using ARCH(13)
  - FUNCTION keyword optional
  - AMODE 64 (64-bit) support
  - Dynamically sized elementary items
  - UTF-8 native data type support
- Miscellaneous changes
  - z/OS versions supported
  - New Reserved Words
  - Predefined compilation variable changes
  - Compiler listing changes
  - PPA changes
  - CEEDUMP changes

Announce/GA dates & offerings

- **Announce**: **September 3, 2019**
  - Note: IBM z15 announcement was Sep 12, 2019, so compiler Announcement letter does not mention z15, and the Enterprise COBOL doc library was not published until Sep 12 since our documentation includes z15 material
- **GA: September 6, 2019**

- There are 3 versions of IBM Enterprise COBOL V6.3:
  - **5655-EC6**: **IBM Enterprise COBOL for z/OS** (MLC based product)
  - **5697-V61**: **IBM Enterprise COBOL Value Unit Edition for z/OS**
    - OTC based pricing (One Time Charge)
    - Product ID unique to ordering process, product registration & SMF Type 1 record generation for sub-capacity reporting
    - Under the covers, it's the 'same product' (ie. same FMIDs/COMPIDs as MLC version) and customers apply same PTFs as MLC version in order to get fixes
  - **5655-TY6**: **IBM Enterprise COBOL Developer Trial for z/OS**
    - 90-day free trial
    - Refreshed with each PTF to keep it up to date

z15 exploitation using ARCH(13)

Changed ARCH compiler option

- ~~ARCH(7) (the default in 6.2)~~
  - 2094-xxx models (IBM System z9 EC) 2096-xxx models (IBM System z9® BC)
- ARCH(8) (the default in 6.3)
  - 2097-xxx models (IBM System z10 EC) 2098-xxx models (IBM System z10 BC)
- ARCH(9)
  - 2817-xxx models (IBM zEnterprise z196 EC) 2818-xxx models (IBM zEnterprise z114 BC)
- ARCH(10)
  - 2827-xxx models (IBM zEnterprise EC12) 2828-xxx models (IBM zEnterprise BC12)
- ARCH(11)
  - 2964-xxx models (IBM z13) 2965-xxx models (IBM z13s)
- **ARCH(12)**
  - **3906-xxx models (IBM z14) 3907-ZR1 models (IBM z14)**
- **ARCH(13)**
  - **8561-xxx models (IBM z15)**

z15 exploitation using ARCH(13)

- Aligned Vector Load/Store Hints
  - Mask bits added to existing vector load and store instructions
  - Compiler can indicate when loads/stores are aligned, possibly allowing hardware to be more efficient
  - Mostly applies in spills (vector load/store) or when saving and restoring vector registers in prolog/epilog code (vector load/store multiple)
- Vector Packed-Decimal Enhancement Facility
  - Mask bit added to existing vector packed instructions to tell hardware to suppress overflow and ignore decimal-overflow mask in the PSW
  - COBOL doesn't have an exception when there's an overflow; COBOL doesn't touch the decimal-overflow mask
  - Other languages (C++, Java) do have exceptions; they set the mask
  - In mixed-language applications, overflows in COBOL code trigger an exception (because the mask is on)
  - LE handles the exception and suppresses for COBOL; lots of overhead
  - New mask bit allows COBOL overflows to be suppressed without LE support; much faster for mixed-language applications

z15 Acceleration for Multi-Language Applications

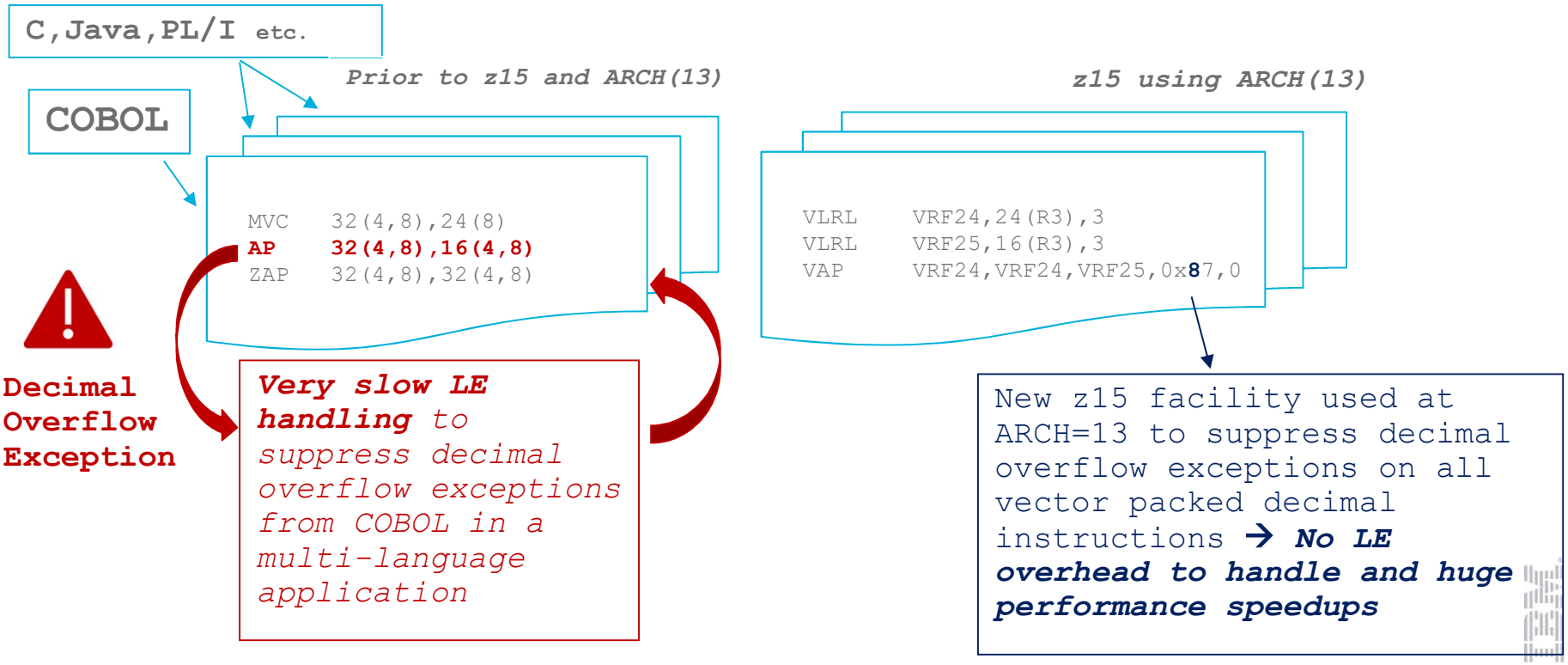
COBOL 6.3 Using ARCH(13)

```
01  WS-VAR-1        PIC S9(5)V9(2) COMP-3.
01  WS-VAR-2        PIC S9(5)V9(2) COMP-3.
01  WS-VAR-3        PIC S9(5)V9(2) COMP-3.

    ADD WS-VAR-1 TO WS-VAR-2 GIVING WS-VAR-3
```

z15 using COBOL 6.3 **ARCH=13** is:

- ✓**100x faster** in a microbenchmark
- ✓**20% faster** in realistic banking simulation multi-language benchmark



z15 exploitation using ARCH(13)

- Vector-Enhancements Facility 2
  - FUNCTION REVERSE is implemented using vector load/store reverse instructions instead of with a runtime call
    - Loop with VLBR to load 16-byte chunks in reverse order
    - Remaining 1-15 bytes reversed with VSTEBR[G|F|H]
  - VECTOR STRING SEARCH used for some INSPECT statements
    - These were already done inline; new code used for narrow cases where there's a performance improvement over existing code
    - Loop with VSTRS
    - INSPECT TALLYING: search string of 6-16 bytes, or 4-16 if a pattern in the string is repeated (e.g. "ABAB")
    - INSPECT REPLACING: search string of exactly 6 bytes
- Miscellaneous Instruction-Extensions-Facility 3
  - SELR/SELGR used for ternary operations
  - Very minor increase; saves a LR (load register) instruction

z15 exploitation using ARCH(13)

New Instructions				
SELGR	SELR	VLBR	VSTEVBF	VSTEVBG
VSTEVBH	VSTRS			
Modified Instructions (mask bits)				
VAP	VCVB	VCVBG	VCVD	VCVDG
VDP	VL	VLM	VLR	VMSP
VPSOP	VRP	VSDP	VSP	VSRP
VST	VSTM			

z15 exploitation using ARCH(13)

COBOL Statements That Will Benefit

Statement	Types	Comments

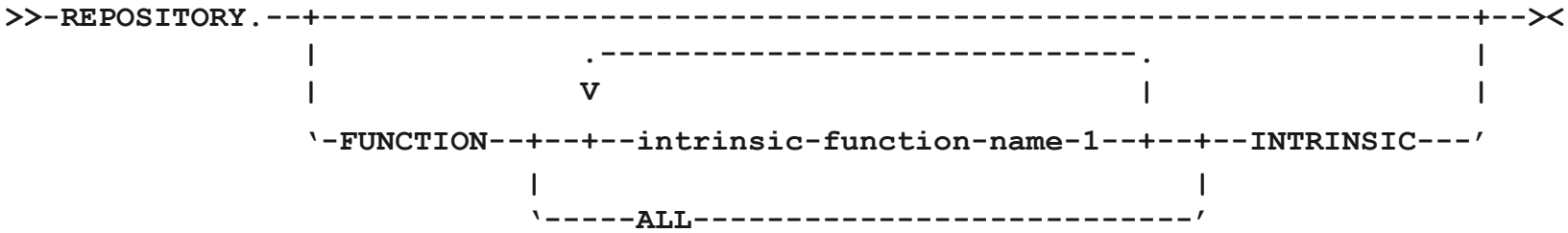
IF X > 0
  Y = 1
ELSE
  Y = 0
	Binary	Ternary (three way) operations are slightly faster

COMPUTE: ADD/SUBTRACT MULTIPLY/DIVIDE REMAINDER	Packed/Zoned	Overflows are just as fast in a mixed-language application as in a pure COBOL application
COMPUTE X = FUNCTION REVERSE(Y)	Alphanumeric	Avoids a runtime call; big performance increase
INSPECT X TALLYING ALL “ABCDEF”	Alphanumeric	Some performance increase over old inlined INSPECT code
INSPECT X REPLACING ALL “ABCDEF” WITH “UVWXYZ”	Alphanumeric	Some performance increase over old inlined INSPECT code

FUNCTION keyword optional

REPOSITORY paragraph, FUNCTION specifier

- The REPOSITORY paragraph FUNCTION specifier INTRINSIC allows declaration of intrinsic-function-names that may be used without specifying the word FUNCTION.
- Syntax:



- *intrinsic-function-name-1*
  - The name of a supported Enterprise COBOL intrinsic function
- ALL
  - If ALL is specified, it is as if each of the supported Enterprise COBOL intrinsic function names were specified

Notes:

1. See the Enterprise COBOL for z/OS V6.2 Language Reference, Intrinsic Functions for a list of supported Intrinsic functions
2. If any *intrinsic-function-name-1* is specified more than once in the REPOSITORY paragraph, all the specifications for that name shall be identical.
3. *Intrinsic-function-name-1* shall not be specified as a user-defined word within the scope of this REPOSITORY paragraph.
4. If ALL is specified, none of the names of the intrinsic functions may be specified as a user-defined word within the scope of this REPOSITORY paragraph.
5. Since WHEN-COMPILED is both a special register and an intrinsic function name, WHEN-COMPILED may not be specified in the FUNCTION clause of the REPOSITORY paragraph.

REPOSITORY paragraph, FUNCTION specifier

```
Environment Division.
Configuration section.

Repository.
    function ABS intrinsic.

Data division.
Working-storage section.
01 int1          pic s9(8) comp value -999.
01 intresult     pic 9(8) comp.

Procedure division.
    compute intresult = ABS(int1)

    if intresult = 999  display "success"
    else                display "fail"
    end-if
    goback.
```

AMODE 64 support

- COBOL finally has AMODE 64!
  - IBM has been working on this for over 10 years
- C/C++ was first, then PL/I, now COBOL
- Some decisions about AMODE 64 support in LE were made 15 years ago
- We are working with other parts of IBM to get support for AMODE 64 programs
  - Debug Tool and CICS are the most important

LP(32|64) compiler option

- Use the LP compiler option to indicate whether an AMODE 31 or AMODE 64 program should be generated with the related language features enabled.

  Syntax: LP(32|64)

  Default: LP(32)

  - When LP(64) is in effect, the compiler generates an AMODE 64 program using the z/Architecture® 64-bit instructions
  - When LP(32) is in effect, the compiler generates an AMODE 31 program.
    - This is the default and is the same as in previous compilers.
- You can specify LP in any of the ways that you specify other compiler options.
  - In a sequence of programs in the same source, if you specify LP in a PROCESS (or CBL) statement it will apply to all programs in the sequence.
- LP(32) and LP(64) are mutually exclusive. If they are specified multiple times, the compiler will take the last one specified:

```
CBL LP(64) LP(32) LP(64)                    *> Will get LP(64)
```

Compiling and binding AMODE 64 programs

- You can compile AMODE 64 COBOL applications under z/OS using job control language (JCL), TSO commands, CLISTs, or ISPF panels. The process is the same as for AMODE 31 applications.
- The key differences are as follows:

```
//COMPILE EXEC PGM=IGYCRCTL,REGION=0M,
//         PARM='LP(64)'                          << add LP(64) option
// ...
//BIND EXEC. PGM=IEWBLINK,REGION=0M,
//         PARM='RENT,DYNAM=DLL                   << add RENT and DYNAM=DLL
//SYSLIB DD. DSN=CEE.SCEEBND2,DISP=SHR            << include SCEEBND2
// ...
//SYSLIN DD ...
//              DD DSN=CEE.SCEELIB(CELQV004),DISP=SHR    << include CELQV004
// ...
```

Compiling and binding AMODE 64 programs

- If an ENTRY statement is used, you must specify ENTRY CELQSTRT
  - If you specify any other ENTRY name, the program will ABEND
  - The entry point now defaults to CELQSTRT, ENTRY not required
  - With LP(32), as in previous compilers, the entry point defaults to the PROGRAM-ID name

- When compiling with JCL, IBM provides a set of cataloged procedures which can reduce the amount of JCL coding that you need to write.
  - IGYQC - A single-step cataloged procedure for compiling a AMODE 64 COBOL program
  - IGYQCB - A two-step cataloged procedure for compiling and binding an AMODE 64 COBOL program
  - IGYQCBG - A three-step cataloged procedure for compiling, binding, and running a AMODE 64 COBOL program

Running AMODE 64 programs

- Mixing AMODE 64 and non-AMODE 64 programs is not supported (neither AMODE 24 nor AMODE 31)
  - This decision was made around 2004 by LE and z/OS
  - We are working on solutions to this!
- When preparing to run an application that contains AMODE 64 COBOL programs, ensure that the SCEERUN2 and SCEERUN Language Environment load libraries are available in the system library search order, for example, by using a STEPLIB DD statement.
- Note that users can compile AMODE 64 COBOL programs on z/OS V2R2 or above, but must be on z/OS V2R3 or above to run AMODE 64 COBOL programs.

Specifying COBOL runtime options

- COBOL-specific runtime options can be specified using the new IGZOPTS DD statement

  Below shows an example of setting the DEBUG option to on:

```
//…
//IGZOPTS     DD      *
  DEBUG
/*
```

  - The option syntax is the same as the corresponding specifications in AMODE 31 using the LE CEEOPTS DD statements.

- When running in z/OS UNIX, you can specify COBOL runtime options using the new *_IGZ_RUNOPTS* environment variable. The option syntax is the same as their corresponding specifications in AMODE 31 using the LE environment variable *_CEE_RUNOPTS*.

  - The example below specifies the DEBUG option:

```
export   _IGZ_RUNOPTS="DEBUG"
```

Specifying COBOL only runtime options (cont.)

- COBOL runtime options for AMODE 64 programs can only be specified using the two methods just described:
  - IGZOPTS DD statements, or
  - *_IGZ_RUNOPTS* environment variable

- The CBLOPTS runtime option is not supported for AMODE 64 programs. When specifying runtime options on the parameter string, runtime options must be specified before program arguments. This behavior is consistent with other Language Environment high- level languages.

The example below shows the JCL for invoking a COBOL main program with program argument '20190901' and runtime option RPTOPTS(ON)

```
//GOSTEP EXEC PGM=COBMAIN,PARM='RPTOPTS(ON)/20190901'
```

AMODE 64 – Runtime Options

- **LE Runtime Options**
  - ALL31: This runtime option is not supported for AMODE 64 programs.
  - MSGFILE: This runtime option is not supported for AMODE 64 programs.
  - CBLPSHPOP: This option affects the behaviour of CICS. It has no effect as CICS does not support AMODE 64 programs
  - XPLINK: Always in effect for AMODE 64 programs, cannot be turned off
  - In order to be consistent with other LE programming languages, runtime options in JCL EXEC statement are specified before the slash (/) character. Program parameters are specified after the slash.
- **COBOL-only runtime options**
  - COBOL only runtime options in AMODE 64 must be specified using IGZOPTS DD JCL statement or _IGZ_RUNOPTS environment variable. Syntax of the options remain the same as in CEEOPTS and _CEE_RUNOPTS.
  - The following COBOL only runtime options are supported when LP(64) is in effect:
    - AIXBLD
    - CBLQDA
    - DEBUG
    - SIMVRD
    - UPSI
  - RTEREUS is not supported

Example. The following JCL fragment specifies AIXBLD and UPSI:

```
//IGZOPTS DD *
  AIXBLD, UPSI(10010000)
/*
//*
```

AMODE 64 – Limitations

- The following are not currently supported in V6.3 for AMODE 64
  - *CICS compiler option*
  - *SQLIMS compiler option*
  - *XML & JSON support*
  - *THREAD option*
  - *Object-oriented COBOL applications*
  - *Interoperability with 31-bit programs*
  - *Db2 programs using the separate Db2 precompiler*
    - The SQL compiler option (integrated SQL coprocessor) *is* supported when LP(64) is in effect

AMODE 64 – Limitations

- The following are not (and will not be) supported for AMODE 64
  - ***ALTER statement***
    - *GOTO without a target is a bad idea!*
  - ***CALL xx USING file-name-1***
    - *Still supported for AMODE 31*
  - ***DATA(24) compiler option***
    - *WORKING-STORAGE will be above the 2GB BAR*
  - ***DLL/NODLL** compiler option has no effect*
    - *LP(64) programs always behave as if DLL was in effect*
  - ***NORENT compiler option** -- must be RENT*
  - ***RMODE(24) compiler option** -- must be RMODE(ANY)*
    - *Note: RMODE 64 not supported (yet?)*
  - ***RTEREUS runtime option***

AMODE 64 – Other considerations (cont.)

➢ Compilation Listing changes

- In the Pseudo Disassembler section of the listing, at the end of a program (CSECT), the text "*Size of dynamic storage*" is changed to "*Size of storage acquired*". For example:

```
***    General purpose registers used: 1111111111111000
***    Floating point registers used: 1100000000000000
***    Vector registers used: 11000000000000000000000000000000
***    Size of storage acquired: 704
***    Size of executable code: 2966
```

- Some of the section headings in the compilation listing are changed in COBOL V6.3.

V6.2 Section Name	V6.3 Section Name
STATIC MAP	INITIAL HEAP STORAGE MAP
WORKING-STORAGE MAP	ABOVE THE BAR HEAP MAP (for LP(64)
WORKING-STORAGE MAP	BELOW THE BAR HEAP MAP (for LP(32))
WSA24 MAP	BELOW THE LINE HEAP MAP
AUTOMATIC MAP	STACK STORAGE MAP

PPA4 and finding WORKING-STORAGE section in 64bit

- The layouts of PPA1, PPA2 and PPA3 remain the same between AMODE 31 and AMODE 64
- There is a new layout for PPA4
  - The information in PPA4 is needed to find the WORKING-STORAGE section
  - The AMODE 64 PPA4 layout is included in the Technical Note for COBOL 6.3 Runtime Enablement PTF
  - It is also in the latest Programming Guide
- Note: Refer to the Technical Note for COBOL 6.3 Runtime Enablement PTF
  A. *SCEELKED and SCEERUN datasets for Enterprise COBOL 6.3*
  B. *Updates in the LE Vendor Interface Guide for* PPA4 layout
  C. *Steps to locate the WORKING-STORAGE section of COBOL V5 and later programs*

AMODE 64 – Storage allocation

- Storage allocation for the following COBOL data types depends on the setting of the LP compiler option. They are 4 bytes when LP(32) is in effect, they are 8 bytes when LP(64) is in effect:
  - USAGE POINTER (also the ADDRESS OF and LENGTH OF special registers, which implicitly have this usage)
  - USAGE FUNCTION-POINTER
  - USAGE OBJECT REFERENCE
  - USAGE INDEX

- If the SYNCHRONIZED clause is specified for a data item that has one of the usages shown above, the item is aligned on a full-word boundary if LP(32) is in effect, or on a double-word boundary if LP(64) is in effect. (This applies to PROCEDURE-POINTER as well).

- Index-names are not data-names they are compiler generated and managed special registers for the use of this object program only. However, there are operations that can be specified between an index-name and a data-name. For example, there can be operations between an index-name and a USAGE INDEX data item. If LP(64) is in effect, index names will be larger than with LP(32).

AMODE 64 – Storage allocation (cont.)

- LENGTH OF special register
  - If LP(32) is in effect, the LENGTH OF special register has this implicit definition:

```
PICTURE 9(9) USAGE IS BINARY
```

  - If LP(64) is in effect, the LENGTH OF special register has this implicit definition:

```
PICTURE 9(18) USAGE IS BINARY
```

- Intrinsic functions:
  - If LP(32) is in effect, the returned value of the LENGTH, UPOS, UVALID, ULENGTH and USUPPLEMENTARY intrinsic function is a 9-digit integer. If LP(64) is in effect, the returned value is an 18-digit integer.

AMODE 64 – POINTER-32 data item

- A data item defined with USAGE IS POINTER-32 is a pointer data item. A POINTER-32 data item can be used in both LP(32) and LP(64) programs. It is a 4-byte elementary item regardless of the LP compiler option setting.

- You can use POINTER-32 data items to accomplish limited base addressing. Pointer data items can be compared for equality or moved to other pointer items.
  - *Note: We will not support PROCEDURE-POINTER-32 and FUNCTION-POINTER-32 as the code CSECTs of Language Environment must reside below the bar.*

- POINTER-32 can be used wherever POINTER data item can be used. POINTER-32 can be set to a POINTER data item and vice versa. POINTER-32 data item can be compared to a POINTER data item.

- When the LP(32) compiler option is in effect, USAGE POINTER and USAGE POINTER-32 are synonyms. A POINTER-32 data item behaves exactly the same as a POINTER data item as-if the POINTER-32 keyword is replaced by the POINTER keyword in the compilation.

AMODE 64 – ALLOCATE statement changes

- The ALLOCATE statement obtains dynamic storage.

Format

```
>>-ALLOCATE--+-arithmetic-expression-1--CHARACTERS-+------------>
             '-data-name-1-------------------------'

>--+-------------+--+----------------+-------------------------->
   '-INITIALIZED-'   '-LOC--integer-1-'

>--+-----------------------+-------------------------------><
   '-RETURNING--data-name-2-'
```

Note:

- The LOC phrase controls how ALLOCATE acquires storage:
  - LOC 24 causes ALLOCATE to acquire storage below the 16MB line
  - LOC 31 causes ALLOCATE to acquire storage below the 2GB bar
  - LOC 64 causes ALLOCATE to acquire storage above the 2GB bar
- **If LOC 64 is specified** and *data-name-2* references a USAGE POINTER-32 data item, a diagnostic message will be issued. The size of the data item is too small for the 64bit address.
- **If the LOC phrase is omitted** and *data-name-2* references a USAGE POINTER data item:
  1. If LP(64) is in effect, the allocated storage will be above the 2 GB bar
  2. Otherwise, the allocated storage will be below the 2 GB bar
- **If the LOC phrase is omitted** and *data-name-2* references a USAGE POINTER-32 data item, the allocated storage will always be below the 2 GB bar regardless of the LP compiler option.

AMODE 64 – FREE statement changes

- The FREE statement releases dynamic storage that was previously obtained with an ALLOCATE statement..

  Format

```
          .--------------.
          V               |
>>-FREE-+--data-name-1-+-----------------------------------><
```

- Additional syntax rules for ***data-name-1*** are as follows:
  - *data-name-1* must be defined as USAGE POINTER or USAGE POINTER-32.

AMODE 64 – CALL statement changes

- The CALL statement transfers control from one program to another within a run unit. In addition to existing syntax rules, the following rules apply in AMODE 64 mode:
  1. Static, Dynamic, and DLL calls can be used to call other AMODE 64 Language Environment conforming programs
  2. AMODE 64 COBOL programs cannot be called by non-Language Environment conforming programs
     - Assembler programs using LOAD and then branch to the entry point of the subprogram will not work. Instead, use the LE macro CEEFETCH and call AMODE 64 COBOL programs
  3. Parameter passing convention is XPLINK
     - The LE runtime option XLPLINK behavior is always in effect

AMODE 64 – Predefined compilation variable IGY-LP

- Predefined compilation variable IGY-LP gives the setting of the LP compiler option
  - *values are 32 or 64*

- The variable can be used in conditional compilation directive as shown in the sample code below:

```
>>IF    IGY-LP = 64
       ALLOCATE  100 CHARACTERS
                 LOC  64
                 RETURNING  WS-P1.
>>END-IF
```

AMODE 64 – Other considerations

- **I/O**
  - QSAM and VSAM files created by AMODE 31 programs can be accessed (read, write and rewrite) by AMODE 64 programs, and vice-versa. Data files are fully compatible between AMODE 64 and AMODE 31 programs.
  - Note: Under the covers, macros provided by DFSMS have to run under AMODE 31/24. The runtime library will handle AMODE switching between the generated code and DFSMS. The record buffer also needs to be below the bar. The runtime library will allocate storage for FD records.

    Code for I/O declarative sections are treated by the compiler as if they are nested subprograms (similar to the handling of SORT INPUT/OUTPUT procedure). This simplifies the transfer of control to the declarative sections. The external behavior behavior of the COBOL program is not affected.

Dynamic-length Data Items

Dynamic-length Data Items - Overview

- A data item where the length may change at runtime depending on the size of the data being moved into it.
- Data description entry syntax for the DYNAMIC LENGTH clause:

```
>>---DYNAMIC---+--------+--+-------------------------+---<<
               |        |  |                         |
               +-LENGTH-+  +-LIMIT-+----+-integer-1-+
                                   |    |
                                   +-IS-+
```

- Example data description entries

```
01 MY-DYN PIC X DYNAMIC.
01 DLEI-02 PIC X DYNAMIC LIMIT 500.
```

- The LENGTH keyword is optional.
- The LIMIT phrase specifies the maximum length of the data item.
  - The data item will not grow past this limit.
  - If a sender is longer than the receiver LIMIT value then the data will be truncated on the right.
  - integer-1 defaults to 999999999 if the LIMIT phrase is not specified.

Dynamic-length Data Items - Overview

- When a dynamic-length elementary item is the receiving operand in a COBOL statement, the compiler generates code to allocate a data buffer at runtime to contain the data. A new data buffer may be allocated if an existing one was not large enough to contain the new data (in which case the old data buffer will be freed).

- Dynamic-length elementary items - general usage rules.
  - They may not be variably-located or located within an OCCURS DEPENDING ON table.
  - They cannot be REDEFINED or RENAMED.
  - They cannot be JUSTIFIED RIGHT.
  - Their address cannot be taken with the ADDRESS-OF special register

Dynamic-length Data Items - Overview

- There are now two kinds of data items: fixed-length items and dynamic-length items. Dynamic-length elementary elementary items and groups that contain dynamic-length elementary items are dynamic-length items. Everything else (including OCCURS DEPENDING ON tables) are fixed-length items.
- Any comparisons involving dynamic-length group items are not yet supported (see continuous delivery)
- When you compare a dynamic-length item with a fixed-length item, the comparison follows the normal comparison rules (extend and pad the shorter operand on the right with spaces, then compare).
- When you compare two dynamic-length elementary items, there are new rules:
  - The lengths are compared first. Depending on the comparison operator (EQ, NEQ, GT, LT, GTE, LTE), the comparison may result in a true or false result before any characters are examined.
  - If the length comparison passes, the characters are examined.

Dynamic-length Data Items - Overview

- Example comparison between two dynamic-length elementary items

```
*> DL-A is length 5, DL-B is length 4
*> Notice a space at the end of DL-A
1 DL-A PIC X DYNAMIC VALUE 'abcd '.
1 DL-B PIC X DYNAMIC VALUE 'abcd'.

IF DL-A = DL-B THEN
  DISPLAY 'DL equal'
ELSE
  DISPLAY 'DL not equal'
END-IF
```

Output is:
DL not equal

Fixed-length Data Items - Review

- Contrast with comparison between two fixed-length elementary items

```
*> FL-A is length 5, FL-B is length 4
*> Notice a space at the end of FL-A
1 FL-A PIC X(5) VALUE 'abcd '.
1 FL-B PIC X(4) VALUE 'abcd'.

IF FL-A = FL-B THEN
  DISPLAY 'FL equal'
ELSE
  DISPLAY `FL not equal'
END-IF
```

Output is:
FL equal

Dynamic-length Items – Available support

- The following are supported for dynamic-length items
  - In the DATA DIVISION
    - Dynamic-length elementary items of class and category alphanumeric, with PICTURE X or PICTURE U.
    - Level 01, 02-49, and level 77.
    - The VALUE clause is allowed.
    - The LIMIT phrase is allowed.
    - Dynamic-length elementary items in an OCCURS table.
      - Not OCCURS DEPENDING ON
    - WORKING-STORAGE and LOCAL-STORAGE sections.
  - In the PROCEDURE DIVISION
    - Reference modification and subscripting.
    - Using dynamic-length elementary items in the MOVE statement as a sender and/or receiver.
    - Using dynamic-length elementary items in relation conditions in IF and EVALUATE statements.

Dynamic-length items – Available support

- Also in the PROCEDURE DIVISION
  - Intrinsic functions are supported except for MIN and MAX
  - The LENGTH OF special register is supported (note: LENGTH OF a dynamic-length item that is a table element is NOT supported)
    - Class conditions are supported
    - The STOP RUN, CANCEL, EXIT PROGRAM, and GOBACK statements are supported
      - In the appropriate context, the compiler will generate code to free any allocated buffers for dynamic-length elementary items with data description entries in the WORKING-STORAGE SECTION.

Dynamic-length Items – Available support

- Also in the PROCEDURE DIVISION
  - The STRING statement is supported.
    - Users may mix-and-match fixed length and dynamic-length items as senders and/or delimiters.
    - The receiver may be a dynamic-length elementary item.
    - The WITH POINTER phrase may point to a character position *outside* of the existing receiver data area, in which case the dynamic-length receiver will be appropriately padded and re-sized. (The pointer position must still be within the dynamic-length receiver LIMIT value).
  - The UNSTRING statement is supported.
    - Users may mix-and-match fixed length and dynamic-length items as receivers and/or delimiters.
    - The sender may be a dynamic-length elementary item.
    - The WITH POINTER phrase may point to a character position *outside* of the existing receiver data area, in which case the dynamic-length receiver will be appropriately padded and re-sized. (The pointer position must still be within the dynamic-length receiver LIMIT value).

Dynamic-length Items – Available support

- The following new syntax is available
  - SET LENGTH OF identifier-1 TO identifier-2
    - identifier-1 must be a dynamic-length elementary item.
    - identifier-2 must be a numeric item or a numeric literal.
    - Allows setting the length of a dynamic-length elementary item.
    - The value of identifier-2 must be greater than or equal to 0, and less than or equal to the value of the dynamic-length elementary item LIMIT.
  - CALL ... USING AS FIXED LENGTH integer-1 phrase
    - Allows passing dynamic-length elementary items as if they were fixed length items of length integer-1. The called program may receive these arguments as fixed-length PIC X(integer-1) items.
    - The AS FIXED LENGTH phrase is only allowed to be specified with dynamic-length elementary items.
    - If the length of the item is shorter than integer-1, then the item will be extended and padded with blanks up to integer-1.
    - After the CALL statement, the length of integer-1 will remain the same as before the call (regardless of the value of integer-1).

Dynamic-length Items - Limitations

- Dynamic-length items are not allowed in the following cases at GA. Work will continue and these items are in plan for subsequent continuous delivery releases.
  - ACCEPT format 1
  - INITIALIZE, SEARCH, INSPECT
  - XML and JSON support
  - Dynamic-length group items in statements
  - PROCEDURE DIVISION USING
  - Dynamic-length structure names in the ENVIRONMENT division (not group names)
  - National (PIC N) with the DYNAMIC-LENGTH clause
  - Intrinsic functions MIN and MAX
  - GLOBAL, EXTERNAL

UTF-8 native data type support

UTF-8 data item support – Overview

- A new USAGE is available: UTF-8
  - USAGE UTF-8 is a new UTF-8 “class” of data as well as a new UTF-8 “category” of data.
- A new picture symbol ‘U’ is introduced that indicates UTF-8 character data.
- Enterprise COBOL treats a single UTF-8 “character” as equivalent to a single Unicode *code point*.
- UTF-8 characters are different from NATIONAL or DBCS characters in that the byte length of each UTF-8 character varies between 1 and 4 bytes.
  - ASCII characters are a subset of UTF-8 and are always encoded in UTF-8 with a single byte.
  - Most commonly used characters can be encoded in UTF-8 with 1-3 bytes (e.g., symbols in the Unicode “Basic Multilingual Plane” (BMP)).

UTF-8 data item support

- Three methods of defining UTF-8 data items will be supported:
- **Method 1:** Fixed character-length UTF-8 items

```
01 u1 PIC U(n) [USAGE UTF-8].   *> USAGE is optional
```

  - This defines a fixed character-length UTF-8 data item that holds exactly n UTF-8 “characters” that occupy between n and 4*n bytes.
    - In move operations, UTF-8 data items are always padded with UTF-8 spaces to a *character* length of n. Truncation is done on UTF-8 character boundaries.
    - NOTE: While 4 * n bytes are always reserved for this type of UTF-8 item, the number of bytes that are actually in use at any one time varies between n and 4*n bytes due to the length of UTF-8 characters (between 1 and 4 bytes each).
      - When a fixed-character length UTF-8 item is used as a receiver in a COBOL statement, unused bytes are always padded with the UTF-8 blank character (x‘20’).

UTF-8 data item support

- **Method 2:** Fixed byte-length UTF-8 items
  - `01 u2 PIC U BYTE-LENGTH [IS] n [USAGE UTF-8].`
    - This defines a fixed byte-length UTF-8 data item that holds exactly n bytes of valid UTF-8 data and <= n UTF-8 characters.
    - When used as receivers in COBOL statements, these UTF-8 data items are always padded with UTF-8 spaces to a byte length of n.
    - Truncation is performed at UTF-8 character boundaries.
  - Fixed byte-length UTF-8 data items are provided for ease of interoperability with Db2, MQ and are strongly recommended to be used when a UTF-8 SORT/MERGE key is needed.

UTF-8 data item support

- **Method 3:** Dynamic length UTF-8 items

```
01 u3 PIC U DYNAMIC LENGTH [LIMIT n] [USAGE UTF-8].
```

  - Defines a dynamic length (dynamically sized) UTF-8 data item
  - These UTF-8 data items have no fixed number of characters and no fixed number of bytes unless the LIMIT clause is specified
  - Dynamic length UTF-8 data items are not padded during MOVE operations
  - Dynamic length UTF-8 data items are only truncated if the specified limit (or max limit for the type) is reached. Truncation is done on UTF-8 character boundaries.

UTF-8 data item support

- Two new types of literals are supported:

- **Type 1:** Basic UTF-8 literals

  - U*'character-data'*
    - *Character-data* is converted from EBCDIC to UTF-8.
    - *Character-data* may contain double-byte EBCDIC characters, but those characters must be delimited by shift-out and shift-in characters.
    - The maximum number of Unicode code points that can be represented in a basic UTF-8 literal can vary depending on the size of each UTF-8 character. However, a maximum of 160 bytes after conversion to UTF-8 is allowed before truncation occurs.

UTF-8 data item support

- **Type 1:** Basic UTF-8 literals (cont.)
  - *Character-data* can contain the following Unicode escape sequences:
    - \u*hhhh*, where each *h* represents a hexadecimal digit in the range '0' to '9', 'a' to 'f' and 'A' to 'F'. Valid range is \u0000 through \uFFFF.
      - This Unicode escape sequence indicates a Unicode code point from the *Basic Multilingual Plane (BMP).*
    - \U*00hhhhhh*, where each *h* represents a hexadecimal digit in the range '0' to '9', 'a' to 'f' and 'A' to 'F'. Valid range is \U00000000 through \U0010FFFF.
      - This Unicode escape sequence can represent any valid Unicode code point, including code points from outside the *Basic Multilingual Plane* (e.g., an emoji symbol).

UTF-8 data item support

- **Type 1:** Basic UTF-8 literals (cont.)
  - *Character-data* can contain the following Unicode escape sequences:
    - Wherever a Unicode escape sequence appears in a basic UTF-8 literal, it is replaced by the compiler with the corresponding UTF-8 encoding for the Unicode code point corresponding to the escape sequence. This makes it convenient to represent general Unicode code points in the literal using only EBCDIC characters.

      e.g., u'caf\u00E9' represents the string 'café'

UTF-8 data item support

- **Type 2:** Hexadecimal UTF-8 literals
  - ux'*hexadecimal-digits*'
    - *hexadecimal-digits* are converted to a sequence of bytes to be used verbatim as the UTF-8 literal value.
    - There is a minimum of 2 hexadecimal digits and up to a maximum of 320 hexadecimal digits allowed.
    - The sequence of bytes is validated to ensure it represents a legal sequence of UTF-8 bytes.

UTF-8 data item support

- UTF-8 move rules
  - The following categories of data items are allowed as senders in a MOVE statement with a category UTF-8 receiver data item
    - UTF-8
    - Alphabetic
    - Alphanumeric
    - Alphanumeric-edited
    - Numeric-edited (incl. usage DISPLAY and usage NATIONAL)
    - National
    - National-edited

  - When a legal sender in a move to UTF-8 is not already of category UTF-8, a conversion of the sender data to UTF-8 is done *automatically* during the move processing

UTF-8 data item support

- UTF-8 move rules (cont.)
  - When a category UTF-8 data item is used as a sender in a MOVE statement, the following category of receivers are permitted:
    - UTF-8
    - National
  - When a category UTF-8 data item is used as a sender in a MOVE statement and the receiver is category NATIONAL, the UTF-8 data item is converted to NATIONAL *automatically* as part of the move processing.
  - Truncation is always done for UTF-8 receivers on character boundaries, so that UTF-8 data items always contain valid UTF-8 data.

UTF-8 data item support

- UTF-8 comparison rules
  - A category UTF-8 data item may be compared to items in the following categories:
    - UTF-8
    - Alphabetic
    - Alphanumeric
    - Alphanumeric-edited
    - Numeric-edited (incl. usage DISPLAY and usage NATIONAL)
    - National
    - National-edited
  - When a compatible non-UTF-8 category item is compared with a category UTF-8 item, a conversion of the non-UTF-8 item to UTF-8 is done *automatically* during the compare processing

UTF-8 data item support

- Statements that accept UTF-8 data items in V6.3:
  - MOVE
  - INITIALIZE
  - IF / EVALUATE
  - SORT / MERGE (UTF-8 data items may be used as sort keys)
  - ALLOCATE / FREE
- The following statements do NOT accept UTF-8 data items:
  - ACCEPT
  - INSPECT
  - STRING
  - UNSTRING
  - JSON GENERATE / PARSE
  - XML GENERATE / PARSE

UTF-8 data item support

- The following intrinsic functions can have UTF-8 arguments:
  - BIT-OF
  - BYTE-LENGTH
  - DISPLAY-OF
  - HEX-OF
  - LENGTH
  - LOWER-CASE
  - NATIONAL-OF
  - TRIM
  - UPOS, ULENGTH, UPPER-CASE, USUBSTR, UVALID, USUPPLEMENTARY

- Figurative constants are supported in UTF-8 moves and compares, except for NULL:
  - SPACE / SPACES ZERO / ZEROES QUOTE / QUOTES
  - LOW-VALUE / LOW-VALUES HIGH-VALUE / HIGH-VALUES

UTF-8 – Documentation

- The following docs have been updated with UTF-8 information.
  - Programming Guide:
    - Chapter 7: Processing data in an international environment -> Processing UTF-8 data using UTF-8 data types
  - Language Reference:
    - Chapter 3. Character-strings: COBOL words and literals -> Literals -> UTF-8 literals
    - Chapter 18. DATA DIVISION data description entry -> USAGE clause -> UTF-8 phrase

COBOL V6.3 Miscellaneous Changes

z/OS versions supported

The following z/OS versions are supported with Enterprise COBOL V6.3

- z/OS 2.2 (no AMODE 64)
- z/OS 2.3
- z/OS 2.4 (GA: Sept 30, 2019)

New Reserved words

The following reserved words were added in Enterprise COBOL V6.3

- LIMIT
- POINTER-24 (potential reserved word)
- POINTER-31 (potential reserved word)
- POINTER-32
- POINTER-64 (potential reserved word)
- UTF-8

Predefined compilation variable changes

- Compilation variables are defined automatically by the compiler and typically reflect things like the value of compiler options (e.g., SQL, SQLIMS, CICS, OPT, etc.) or things like the compiler version. They are read-only.
- IGY-LP was added in V6.3
- IGY- was added to each of the other predefined compilation variables in V6.3
  - Old variables without the *IGY-* prefix are tolerated, but it is suggested that the *IGY-* prefix be used when you defining the compilation variables to help identify Enterprise COBOL specific variables

Predefined compilation variable changes

Predefined compilation variable name	Value
IGY-ARCH	The value of the ARCH option that was used to compile the program: 8, 9, 10, 11, 12 or 13.
IGY-CICS	B'1' if CICS compiler option specified; B'0' otherwise.
IGY-COMPILER-VRM	An integer in the format VVRRMM, where: • VV represents the version number. • RR represents the release number. • MM represents the modification number. For example, compiler version 6.3.0 has a COMPILER-VRM value of 060300.
IGY-DLL	B'1' if DLL compiler option is specified; B'0' otherwise.
IGY-DYNAM	B'1' if DYNAM compiler option is specified; B'0' otherwise.
IGY-LP	The value of the LP option that is used to compile the program: 32 or 64.
IGY-OPTIMIZE	Numeric value representing the value of the OPTIMIZE compiler option.
IGY-SQL	B'1' if SQL compiler option is specified; B'0' otherwise.
IGY-SQLIMS	B'1' if SQLIMS compiler option is specified; B'0' otherwise.
IGY-THREAD	B'1' if THREAD compiler option is specified; B'0' otherwise.

Compiler Listing changes

- In the Pseudo Disassembler section of the listing, at the end of a program (CSECT), the text "*Size of dynamic storage*" is changed to "*Size of storage acquired*". For example:

```
***    General purpose registers used: 1111111111111000
***    Floating point registers used: 1100000000000000
***    Vector registers used: 11000000000000000000000000000000
***    Size of storage acquired: 704
***    Size of executable code: 2966
```

- Some of the section heading in the compilation listing is changed in COBOL V6.3. This affects LP(64) and LP(32).

Existing Section Name	V6.3 Section Name
STATIC MAP	INITIAL HEAP STORAGE MAP
WORKING-STORAGE MAP	ABOVE THE BAR HEAP MAP (for LP(64)
WORKING-STORAGE MAP	BELOW THE BAR HEAP MAP (for LP(32))
WSA24 MAP	BELOW THE LINE HEAP MAP
AUTOMATIC MAP	STACK STORAGE MAP

CEEDUMP changes

Existing Section Name	V6.3 Section Name
STATIC MAP	INITIAL HEAP STORAGE MAP
WORKING-STORAGE MAP	ABOVE THE BAR HEAP MAP (for LP(64)
WORKING-STORAGE MAP	BELOW THE BAR HEAP MAP (for LP(32))
WSA24 MAP	BELOW THE LINE HEAP MAP
AUTOMATIC MAP	STACK STORAGE MAP

6.2

```
*****  S T A T I C    M A P  *****

OFFSET (HEX)   LENGTH (HEX)   NAME

           0              4      BLL_Ptrs
           4              C      BLT_Ptrs
          10             60      GPCB
          70              4      WS-BASE-ADDRESS


*****  E N D    O F    S T A T I C    M A P

*****  W O R K I N G - S T O R A G E   M A P

OFFSET (HEX)   LENGTH (HEX)   NAME

           0              4      JNIENVPTR
           8              2      RETURN-CODE
          10              2      SORT-RETURN
          18              8      SORT-CONTROL
          20              4      SORT-CORE-SIZE
          28              4      SORT-FILE-SIZE
          30              4      SORT-MODE-SIZE
          38              8      SORT-MESSAGE
          40              4      TALLY
          48              1      SHIFT-OUT
          50              1      SHIFT-IN
          58              4      XML-CODE
          60             1E      XML-EVENT
          80              4      XML-INFORMATION
          88              4      JSON-CODE
```

6.3

```
*****  I N I T I A L   H E A P   S T O R A G E   M A P  ***

T (HEX)   LENGTH (HEX)   NAME

           0              4      BLL_Ptrs
           4              C      BLT_Ptrs
          10             74      GPCB
          84              4      WS-BASE-ADDRESS


*****  E N D    O F    I N I T I A L   H E A P   S T O R A G E

*****  B E L O W   T H E   B A R   H E A P   M A P  *****

T (HEX)   LENGTH (HEX)   NAME

           0              4      JNIENVPTR
           8              2      RETURN-CODE
          10              2      SORT-RETURN
          18              8      SORT-CONTROL
          20              4      SORT-CORE-SIZE
          28              4      SORT-FILE-SIZE
          30              4      SORT-MODE-SIZE
          38              8      SORT-MESSAGE
          40              4      TALLY
          48              1      SHIFT-OUT
          50              1      SHIFT-IN
          58              4      XML-CODE
          60             1E      XML-EVENT
          80              4      XML-INFORMATION
          88              4      JSON-CODE
```

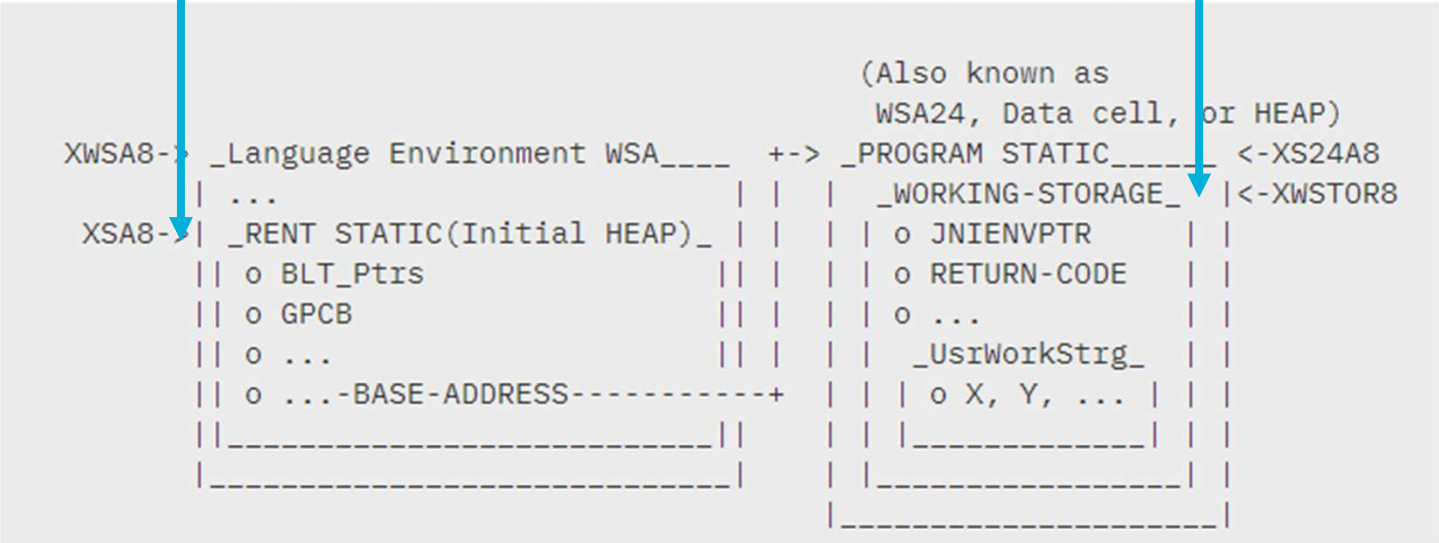
CEEDUMP changes

Existing Section Name	V6.3 Section Name
STATIC MAP	INITIAL HEAP STORAGE MAP
WORKING-STORAGE MAP	ABOVE THE BAR HEAP MAP (for LP(64)
WORKING-STORAGE MAP	BELOW THE BAR HEAP MAP (for LP(32))
WSA24 MAP	BELOW THE LINE HEAP MAP
AUTOMATIC MAP	STACK STORAGE MAP

where are they located and fit together? – Refer to the recent updates in LE Vendor interface

https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.4.0/com.ibm.zos.v2r4.ceev100/IGZXAPI.htm

Figure 1. Layout of the COBOL V5+ RENT program (when DATA(24) or WSOPT bit is ON)



Note: PROGRAM STATIC and WORKING-STORAGE are the same.

CEEDUMP changes – With PH15116 Applied

Existing Section Name	V6.3 Section Name
STATIC MAP	INITIAL HEAP STORAGE MAP
WORKING-STORAGE MAP	ABOVE THE BAR HEAP MAP (for LP(64)
WORKING-STORAGE MAP	BELOW THE BAR HEAP MAP (for LP(32))
WSA24 MAP	BELOW THE LINE HEAP MAP
AUTOMATIC MAP	STACK STORAGE MAP

CEEDUMP (6.3 64bit CEEDUMP can be still buggy and needs to be improved)

```
  +0020 27000720    Inaccessible storage.
Local Variables:
  X1 was not compiled with the SYM suboption of the TEST option.  A formatted variable dump cannot be produced.

  DSA for X1: 26DB3030
  WSA for X1: 26DD3220
  RENT STATIC (Initial HEAP) for X1: 26DD3308

  Note: 'NORENT/RENT STATIC (Initial HEAP)' address corresponds to 'STATIC MAP' in the COBOL compilation listing.
        Use the offset in the listing to locate a variable.
```

```
WSA for X1: 26DD3220
RENT STATIC (Initial HEAP) for X1: 26DD3230
  +000000 26DD3230  00000000 26DD3234 26DD3238 26DD323C  C7D7C3C2 00000001 26DD328C 2690C524  |.............GPCB...
  +000020 26DD3250  00000002 14800000 26DB02A0 00000000  00000000 26DAF430 00000000 00000000  |.....................4
  +000040 26DD3270  00000000 E2E8E2D6 E4E34040 26DAF3B8  F1F9F2F3 F3F6F3F0 00000028 00000001  |....SYSOUT   ..3.1923363
  +000060 26DD3290  81000000 2690C000 00000000 00000000  00000000 00000000 26DD3240 00000000  |a.....................
  +000080 26DD32B0  00000000 26DD3308 00000000 00000000  00000000 00000000 00000000 00000000  |....................

PROGRAM STATIC (WSA24, Data cell, HEAP) for X1: 26DD3308
  +000000 26DD3308  00000000 00000000 00000000 00000000  00000000 00000000 C9C7E9E2 D9E3C3C4  |....................
  +000020 26DD3328  00000000 00000000 00000000 00000000  00000000 00000000 E2E8E2D6 E4E34040  |....................
```

z/OS Academy: June 15-19, 2020
September 21-25, 2020

- On-site event in Poughkeepsie, NY with lectures, demos, and hands-on instruction
- Designed for z/OS System Programmers with 2 – 5 years experience
- Networking opportunities with z/OS developers
- Eligible for **z/OS System Programmer – Advanced** digital badge

FREE!

You need to just cover travel expenses

To Register, please email
David Raften at Raften@us.ibm.com
Ryan Rawlins at RRawlins@us.ibm.com

Topics may include:

Pervasive Encryption
Zowe™
z/OS Upgrade
Cloud Provisioning
SMP/E
z/OSMF
Analytics
zEscape Room
Troubleshooting
Tours
And more selected topics!

Resources

Resources

- COBOL Marketplace and Documentation Library
  - Enterprise COBOL product pages have been replaced by Marketplace pages: https://www.ibm.com/us-en/marketplace/ibm-cobol
  - Enterprise COBOL Documentation Library: http://www-01.ibm.com/support/docview.wss?uid=swg27036733
- Enterprise COBOL for z/OS V6.3 Fact Sheet
  - http://www-01.ibm.com/support/docview.wss?uid=swg22004707
- Enterprise COBOL Migration Assistant
  - https://cobol-migration-assistant.mybluemix.net/
- COBOL Portal on CTWEB: https://ctweb.torolab.ibm.com/wiki/index.php/COBOL_Portal
- Test & Service wiki page (link from the CTWEB COBOL Portal): https://ctweb.torolab.ibm.com/wiki/index.php/Enterprise_COBOL_Test_%26_Service
  - These slides will end up there too

Questions?

AMODE 64 – Documentation

- The following docs have been updated with AMODE 64 (64-bit) information.
  - Programming Guide:
    - Chapter 17: Compiling, binding and running COBOL AMODE 64 applications
    - Chapter 18: Compiler options -> LP
    - Chapter 25: Developing AMODE 64 programs
  - Language Reference:
    - Chapter 18. DATA DIVISION data description entry -> USAGE clause -> POINTER-32 phrase
    - Appendix B. Compiler Limits

Dynamic Length Elementary Items – Documentation

- The following docs have been updated with Dynamic Length Elementary Item information.
  - Language Reference:
    - Chapter 16. DATA DIVISION overview -> Data relationships -> Dynamic-length items
    - Chapter 18. DATA DIVISION data description entry -> DYNAMIC LENGTH clause