Cross Program Communication in z/OS

The following terms that may appear in these course materials are trademarks or registered trademarks:

Trademarks of the International Business Machines Corporation:

AIX, BookManager,CICS, DB2, DRDA, DS8000, ESCON, FICON, HiperSockets, IBM, ibm.com, IMS, Language Environment, MQSeries, MVS, NetView, OS/400, POWER7, PR/SM, Processor Resource / Systems Manager, OS/390, OS/400, Parallel Sysplex, QMF, RACF, Redbooks, RMF, RS/6000, SOMobjects, S/390, System z, System z9, System z10, VisualAge, VTAM, WebSphere, z/OS, z/VM, z/VSE, z/Architecture, zEnterprise, zSeries, z9, z10

Trademarks of Microsoft Corp.: Microsoft, Windows

Trademarks of Micro Focus Corp.: Micro Focus

Trademark of American National Standards Institute: ANSI

Trademarks of America Online, Inc.: America Online, AOL

Trademarks of Quercus Systems: Personal REXX, REXXTERM

Trademark of Chicago-Soft, Ltd: MVS/QuickRef

Trademark of Phoenix Software International: (E)JES

Trademark of Triangle Systems: IOF

Trademarl of Syncsort Corp.: SyncSort

Trademark of CA: Endevor

Trademark of Serena Software International: ChangeMan

Registered Trademarks of Institute of Electrical and Electronic Engineers: IEEE, POSIX

Registered Trademarks of Corel Corporation: Corel, CorelDRAW, Corel VENTURA

Registered Trademark of Oracle Corporation: Oracle

Registered Trademark of The Open Group: UNIX

Trademarks of Sun Microsystems, Inc.: Java, EmbeddedJava, Enterprise JavaBeans, EJB, Java Naming and Directory Interface, JavaBeans, JavaOS, JavaScript, JavaServer, JavaServerPages, JSP, JDBC, JDK, JVM, J2EE, Sun Microsystems, 100% Pure Java

Registered Trademark of Linus Torvalds: LINUX

Registered Trademark of Unicode, Inc.: Unicode

Trademarks held on behalf of World Wide Web Consortium: W3C, XHTML, XSL, WebFonts

Trademark of Object Management Group: CORBA

Trademarks of Apple Computer: QuickTime, Safari

Trademarks of Adobe Systems, Inc.: Macromedia, PDF, Shockwave, Flash

Trademark of The Eclipse Foundation: Eclipse

Cross Program Communication in z/OS - Course Objectives

On successful completion of this class, the student, with the aid of the appropriate reference materials, should be able to:

1. Code calling and called programs using one or more of these compilers:
   * Enterprise COBOL
   * XL C/C++ for z/OS
   * Enterprise PL/I

   or * High Level ASseMbler (HLASM) language

2. Define elementary and aggregate data types in all of these languages
3. Access JCL PARM data from a main program written in any of these languages, and set the JCL return code value; access the parm data from a subroutine written in any of these languages using the CEE3PRM or CEE3PR2 services
4. Describe the general content of object modules in OBJ, XOBJ, and GOFF formats
5. Call subroutines / external functions from each of these languages, statically and dynamically, passing elementary and aggregate data items, passing by reference, by content, and by value, and examining any returned value from the subroutine, as possible for each language
6. Code subroutines in each of these languages, receiving data as it is passed and passing back a return value as appropriate and possible, with an objective of creating subroutines that can be called from programs written in any of the four languages discussed here
7. Describe how argument lists are built and how parameter lists are received in all four languages
8. Use the program binder to create load modules and program objects
9. Create and use programs with multiple entry points
10. Deal with variable numbers of arguments and parameters, as appropriate to each language, and setting and recognizing omitted parameters where possible
11. Where possible, share external data items across programs, modules, and languages.

Cross Program Communication in z/OS - Topical Outline

Day One

Cross Program Communication in z/OS - Topical Outline, p.2.

Day Two

Cross Program Communication in z/OS - Topical Outline, p.3.

Day Three

Languages Selection

- ❒ This course is multi-lingual, but we don't talk about programming languages you will not be encountering

- ❒ So here is the time for you to specify which languages you are interested in exploring during this class

- ❒ Based on your selection(s) we will omit parts of lecture and labs that are not relevant to your work

Language

______ Assembler

______ C

______ COBOL

______ PL/I

This page intentionally left almost blank.

Section Preview

❒ **Introduction to the class**

- ♦ **Interesting Applications**
- ♦ **Coding Notes For Examples in the Class**
- ♦ **Setting Up for the Labs (Machine Exercise)**

Interesting Applications

- Applications that are simple can be written as self-contained single programs as an on-line transaction or a batch job-step

- But interesting (read: complex) applications often need to be written as a mainline (driver) program with one or more subroutines
  - The mainline calls subroutines as needed
  - And subroutines can in turn call other subroutines

- A good design point is to compartmentalize each subroutine to perform a single function
  - If that function can be broken down into sub-pieces, put those pieces into separate subroutines
  - This way, updates and maintenance are localized and simplified

Interesting Applications, 2

- **Typically when a program (mainline or subroutine) calls a subroutine, the caller passes data to the callee**

  - **The called program then accesses the passed data, and may change the passed data**

  - **The called program may also return a value to the caller**

- **Life is sweet and simple if all programs are written in a single language**

  - **But this is often not the case:**

    - ✗ High level language programs, written in COBOL or PL/I, say, may need to call subroutines that were written in Assembler to accomplish some function that cannot be done in the high level language

    - ✗ Conversely, many functions are accomplished more simply in a high level language than in Assembler

    - ✗ Certain computations may be done more naturally in PL/I or C (engineering applications often need to work with math functions and imaginary numbers, for example, tasks not well suited to COBOL)

    - ✗ The person writing the subroutine may prefer to code in a particular language that is not the same as the language of the calling program

Interesting Applications, 3

- **In this class we explore the mysteries and details of coding applications written using external subroutines**

- **This includes programs written in these languages**

  - **Assembler**
  - **COBOL**
  - **PL/I**
  - **C**

- **We examine invoking external routines written in the same language as the invoker and invoking routines written in different languages from the invoker**

- **We are specifically focused on the most current compilers and running in the z/OS LE environment**

  - **We assume you are proficient in at least one of the four languages discussed, but that you may not be familiar with how to work in all of them**

    - So we have provided enough details and clues to enable you to succeed in the labs that use languages that you might not be fluent in

Interesting Applications, 4

❒ **In this class we will explore ...**

- **Formats of data items inherent to z-series machines and how to declare them in the different languages**
  - ✗ Character string
  - ✗ Binary
  - ✗ Packed decimal
  - ✗ Floating point
- **Formats of aggregates**
  - ✗ Structures
  - ✗ Arrays
- **Other data types**
  - ✗ Null-terminated strings
  - ✗ Pointers / addresses
- **Common (External) data**

Interesting Applications, 5

❒ **In this class we also explore ...**

- **How to access the PARM field from the EXEC statement that invokes a main program**

- **How to set a return code that is passed back to z/OS**

- **How to invoke subroutines**

  - ✗ Syntax of call / function reference in multiple languages (Assembler, COBOL, PL/I, C)

  - ✗ Ways to pass data

  - ✗ Issues of static versus dynamic calls

  - ✗ How to access a value returned from a subroutine

- **How to code subroutines**

  - ✗ Ways to catch data

  - ✗ When you can and cannot change passed data

  - ✗ How to pass back a return value

  - ✗ How to code subroutines so that they are callable from all the languages being discussed

Interesting Applications, 6

- We also explore related issues of subroutines
  - Object code structure and components
  - Generalized Object Format (GOFF)
  - ENTRY statements in source
  - Executable module structure and components
  - Program objects
  - How the program binder works
  - Module attributes
  - Using LE and z/OS UNIX services to invoke subroutines
- What we don't cover (but allude to here and there):
  - Multi-tasking, multi-threading
  - XPLINK

Coding Notes For Examples in the Class

- We assume you are using the most recent versions of compilers, the Assembler, z/OS, Language Environment, and the program binder

  - However, most of the discussion is relevant to earlier versions of each of these products

  - Newer versions of these products will be available from time to time and it's good to stay current in your reading

  - Where it is especially critical, versions and levels of products will be specified

- We are concerned with having lots of correct coding examples

  - And we want them to be complete enough for you to use these examples as models / starting points back on the job

  - But, we do not want to clutter up examples with lines of code that should be clear to experienced programmers

  - For example, we will not show declarations of data items unless it is necessary for clarity

  - To simplify the examples, therefore, we have put on these following pages assumptions you can make about unshown segments of a program

Coding Notes For Examples in the Class - Assembler

- **In Assembler examples, <u>we will not show standard save area linkage code</u> unless it is required to demonstrate some aspect of the example**

  - **<u>We will not show the LE Assembler macros</u>, but we will specify if an Assembler example is LE conforming or not, if it makes a difference in behavior**

  - **Generally speaking, everything discussed here works for LE-conforming Assembler, while non-LE conforming Assembler can:**

    - ✗ Call LE COBOL subroutines directly with a lot of overhead or call intermediate routines to first establish the LE environment

    - ✗ Call LE PL/I subroutines only using intermediate routines to first establish the LE environment

    - ✗ Call LE C subroutines only using intermediate routines to first establish the LE environment

- **We <u>will not necessarily show the target of branch instructions</u>, if the content of the code is not central to the example**

- **The following data names may be used in examples, assuming definitions as shown:**

```
fc       dc    12x'00'       for LE feedback
dest     dc    f'2'          for LE message routing
dblwrd   dc    d'0'          for conversions
```

Coding Notes For Examples in the Class - COBOL

- In COBOL examples, we will not show any divisions not necessary for understanding of an example

  - We assume familiarity with COBOL program structure

- We will not necessarily show the target of "perform" statements, if the content of the code is not central to the example

- The following data names may be used in examples, assuming definitions as shown:

```
01  fc     pic x(12)  value low-values.
01  dest   pic s9(9)  binary value 2.
```

Coding Notes For Examples in the Class - PL/I

❒ In PL/I examples, <u>we will not show any code not necessary for understanding of an example</u>

- We assume familiarity with PL/I program structure
- We will not generally show declarations for builtin functions nor LE service routines

❒ We <u>will not necessarily show the target of "call" statements</u>, if the content of the code is not central to the example

❒ The following data names may be used in examples, assuming definitions as shown:

```
dcl  fc    char(12)          init(low(12));
dcl  dest  fixed  binary(31) init(2);
```

❒ There are lots of special cases and options in PL/I not covered here (for example, constructs such as unions and passing arrays that are not CONNECTED)

- But we do cover the vast majority of real world arguments and parameters
- Similar remarks apply to C ...

Coding Notes For Examples in the Class - C

❐ **In C examples, <u>we will not show any code not necessary for understanding of an example</u>**

- **We assume familiarity with C program structure**
- **All C examples may or may not also apply to C++**
- **We will not generally show all #includes, unless necessary to demonstrate some aspect of the example; you need to ensure you have all necessary #include statements in any code you write; be sure to check these:**
  - ✗ #include <leawi.h> for LE services support
  - ✗ #include <decimal.h> for packed decimal support

❐ **We <u>will not necessarily show the target of function references</u>, if the content of the code is not central to the example**

❐ **Examples use standard C notations; but actual code in the labs uses trigraphs, mostly: "??(" for "[" and "??)" for "]"**

❐ **The following data names may be used in examples, assuming definitions as shown:**

```
_FEEDBACK  fc;
long int  dest = 2;
long int i;
long int j;
long int k;
```

Computer Exercise: Setting Up for the Labs

This machine exercise is designed to provide setup for all the remaining class exercises.

First, you need to run M520STRT, a supplied REXX exec that will prompt you for the high level qualifier (HLQ) you want to use for your data set names; the exec uses a default of your TSO id, and that is usually fine. Then the exec creates data sets and copies members you will need.

From ISPF option 6, on the command line enter:

```
===> ex '__________.train.library(m520strt)' exec
```

A panel displays for you to specify the HLQ for your data sets, with your TSO id already filled in. Press <Enter> and you get a panel telling you setup has been successful. Press <Enter> again and you are back to the ISPF command panel.

The allocated data sets:

<hlq>.TR.CNTL	for all your JCL
<hlq>.TR.COBOL	for all COBOL source code
<hlq>.TR.SOURCE	for all other source code
<hlq>.TR.LOAD	for load modules
<hlq>.TR.PDSE	for program objects (if supported in your shop)

This page intentionally left almost blank.

Section Preview

❒ **Defining Elementary Data Items**

- **General Concerns**

- **Data Types - zSeries Hardware**

- **Data Types**
  - ✗ Character String, and code pages
  - ✗ Packed Decimal
  - ✗ Binary Integer - halfword, fullword, doubleword
  - ✗ Floating Point - short, long, extended
  - ✗ Addresses / Pointers
  - ✗ Other Data Types
    - ➢ Edited strings
    - ➢ Bit strings
    - ➢ Null terminated strings

- **Working With Null Terminated Strings**

- **Rules for Names**

- **Defining Elementary Items (Machine Exercise)**

General Concerns

- We begin our discussion with an examination of data types
  - What data types are inherent in the hardware
  - How does each language specify those data types
  - What data types are specific to particular languages

- We discuss each elementary data type and how to define an item of the type in each of the languages we are concerned with
  - Including an example of an initialized item and an uninitialized item

- Note that we do not discuss issues of 64-bit addressability except in the most tangential ways
  - The issues surrounding 64-bit addressability deserve their own discussion

Data Types - zSeries Hardware

❒ **The zSeries class hardware works with these data types**

- **Character string of specific, fixed length**
  - ✗ Encoded in EBCDIC, ASCII, or Unicode
- **Packed decimal data of specific, fixed length (1 to 16 bytes possible)**
- **Binary integer data**
  - ✗ Halfword - two bytes
  - ✗ Fullword - four bytes
  - ✗ Doubleword - eight bytes (zSeries machines)
- **Floating point data, in hexadecimal floating point, binary floating point (IEEE) formats (also, decimal floating point, introduced with z9 machines and z/OS 1.8; this is not discussed in this course)**
  - ✗ Short floating point - four bytes
  - ✗ Long floating point - eight bytes
  - ✗ Extended floating point - sixteen bytes
- **Addresses (pointers) - four bytes (in 24-bit and 31-bit addressing modes) or eight bytes (in 64-bit addressing mode)**

Data Types - Character String

- ❐ **A series of consecutive bytes in memory, containing any data, length is determined by application designer**

Language	Defining characteristics	Examples
Assembler	**DC or DS instruction with data type 'C', possibly explicit length, and for DC an explicit value**	`Ty_fld   DC    C'J2'` `TransCd  ds    cl4`

COBOL	**PIC clause including at least one A or X, possibly with a Value clause (USAGE is implicitly DISPLAY)**	`01  Ty-fld  pic xx`
		`          value 'J2'.`
		`01  TransCd  pic x(4).`

PL/I	**DECLARE of type CHAR, possibly with an INIT clause**	`dcl  Ty_fld  char(2)`
		`         init('J2');`
		`Dcl  TransCd char(4);`

C/C++	**define as a char array; (null-terminated strings discussed later); initial value done by assignment (=) or some strcpy or memcpy type function**	`char Ty_fld [2] = "J2";` `char  TransCd [4];`

Data Types - Character String, 2

❒ **Generally speaking, character strings are just strings of bits**

- **The assignment of the bits to characters is specified by the <u>codepage</u> currently in use**

- **By default, mainframe programs use EBCDIC (Extended Binary Code for Decimal Interchange Characters)**

  - ✗ There are many alternate EBCDIC codepages, depending if you need characters from various languages

- **In modern systems, you may send and receive data that is encoded using other schemes**

  - ✗ Most commonly ASCII (American Standard Code for Information Interchange) or its international counterpart ISCII (International Standard Code for Information Interchange)

  - ✗ While a growing number of applications use Unicode, in one of its three formats (UTF-8, UTF-16, and UTF-32) since Unicode support is required for HTML 4.0, XML, Java, Web Services, and other recent technologies

    - ➢ UTF stands for "Uniform Transformation Code"

  - ✗ An older encoding scheme that most IBM products support is called Double Byte Character Set (DBCS), but this seems to be fading in interest

❒ **Discussion of codepages is beyond the scope of this course, but an awareness of codepage issues is important for modern applications**

Data Types - Character String, 3

- zSeries hardware has instructions added to compare, pack, unpack, move, and otherwise work with Unicode data

  - And some to work with ASCII

  - And some to convert between various encodings

- The language products under discussion also support various codepage work

  - ✗ But we leave that discussion for our course on internationalization

Data Types - Packed Decimal

❒ **A series of consecutive bytes in memory, containing two decimal digits in each byte, except the last hex digit is the sign (Hex A-F)**

Language	Defining characteristics	Examples

Assembler	**DC or DS instruction with data type 'P', possibly explicit length, and for DC an explicit value**	`Amount   DC   p'35.50'`
		
		`Tax      DS   PL4`

COBOL	**PIC clause including one or more 9s, possibly a V (for decimal location), possibly with a Value clause, and a USAGE of PACKED-DECIMAL, COMPUTATIONAL-3, or COMP-3**	`01  Amount  pic S999v99`
		`                comp-3`
		`            value +35.50.`
		
		`01  Tax  pic s9(5)v99`
		`         packed-decimal.`

PL/I	**DECLARE of type FIXED DECIMAL(m,n), possibly with an INIT clause**	`dcl  Amount`
		`     fixed decimal(5,2)`
		`        init(35.50);`
		
		`Dcl Tax dec fixed(7,2);`

C/C++	**Not inherent in C, but for z/OS, include the decimal.h header then define as decimal(m,n); initial value done by assignment**	`#include <decimal.h>`
		`.`
		`.`
		`.`
		`decimal(5,2)  Amount`
		`      = 35.50d;`
		
		`decimal(7,2) Tax;`

Data Types - Binary Integer, halfword

❒ **Two bytes, halfword aligned, containing a binary number**

Language	Defining characteristics	Examples
Assembler	**DC or DS instruction with data type 'H', and for DC an explicit value**	`Counter DC   H'0'` `No_rcs  DS   h`
COBOL	**PIC clause including 1-4 9s, possibly with an S (for sign), possibly with a Value clause, and a USAGE of COMP, COMPUTATIONAL, COMP-4, COMPUTATIONAL-4, COMP-5, COMPUTATIONAL-5, or BINARY**	`01  Counter  pic S9999 binary value +0.` `01  No-rcs   pic s9(4) comp.`
PL/I	**DECLARE of type FIXED BINARY(15), possibly with an INIT clause**	`dcl  Counter fixed binary(15) init(0);` `Dcl No_rcs bin fixed(15);`
C/C++	**declare as** short int, short, signed short, signed short int, unsigned short int, **or** unsigned short**; any initial value comes from an assignment**	`short int Counter = 0;` `signed short No_rcs;`

❒ **Also note that PL/I, C, and Assembler can work with a one-byte binary field, even though that is not a native hardware construct**

Data Types - Binary Integer, fullword

❒ **Four bytes, fullword aligned, containing a binary number**

Language	Defining characteristics	Examples
Assembler	**DC or DS instruction with data type 'F', and for DC an explicit value**	`Quantity DC   F'0'` `No_ents  DS   f`
COBOL	**PIC clause including 5-9 9s, possibly with an S (for sign), possibly with a Value clause, and a USAGE of COMP, COMPUTATIONAL, COMP-4, COMPUTATIONAL-4, COMP-5, COMPUTATIONAL-5, or BINARY**	`01  Quantity  pic S9(9) binary value +0.` `01  No-ents   pic s9(9) comp.`
PL/I	**DECLARE of type FIXED BINARY(31), possibly with an INIT clause**	`dcl  Quantity fixed binary(31) init(0);` `Dcl No_ents bin fixed(31);`
C/C++	**declare as** int, long, long int, signed long, signed int, signed long int, unsigned int, unsigned, unsigned long int, **or** unsigned long; **any initial value comes from an assignment**	`int Quantity = 0;` `signed int No_ents;`

Data Types - Binary Integer, doubleword

❒ **Eight bytes, doubleword aligned, containing a binary number**

Language	Defining characteristics	Examples
Assembler	**DC or DS instruction with data type 'FD', and for DC an explicit value**	`Quantity DC   FD'0'` `No_ents  DS   fd`
COBOL	**PIC clause including 10-18 9s, possibly with an S (for sign), possibly with a Value clause, and a USAGE of COMP, COMPUTATIONAL, COMP-4, COMPUTATIONAL-4, COMP-5, COMPUTATIONAL-5, or BINARY**	`01  Quantity pic S9(18) binary value +0.` `01  No-ents  pic s9(18) comp.`
PL/I	**DECLARE of type FIXED BINARY(63), possibly with an INIT clause**	`dcl  Quantity fixed binary(63) init(0);` `Dcl No_ents bin fixed(63);`
C/C++	**declare as <u>long long</u>; any initial value comes from an assignment**	`long long Quantity = 0;` `signed long long No_ents;`

Data Types - Floating Point, short

- Four bytes, fullword aligned, containing a short floating point number in hexadecimal floating point (HFP) or binary floating point (BFP, the IEEE standard) format

Language	Defining characteristics	Examples
Assembler	**DC or DS instruction with data type 'E' or 'EB', respectively, and for DC an explicit value (decimal or with exponent)**	`Distance  DC   E'5.25'` `In_D      DS   E`

COBOL	**No PIC clause, possibly a Value clause, and a USAGE of COMP-1 or COMPUTATIONAL-1; BFP is not supported**	`01  Distance   comp-1`
		`         value 5.25E1.`
		`01  In-D       comp-1.`

PL/I	**DECLARE of type FLOAT DECIMAL(6) or FLOAT BINARY(21), possibly with INIT; compiler option DEFAULT(IEEE) makes all floating point BFP**	`dcl  Distance`
		`       float decimal(6)`
		`       init(5.25E1);`
		`Dcl In_D dec float(6);`

C/C++	**declare as float (decimal or with exponent); any initial value done by assignment; compiler option FLOAT(IEEE) makes all floating point items BFP**	`float  Distance = 5.25;` `float In_D;`

Data Types - Floating Point, long

- ❒ **Eight bytes, doubleword aligned, containing a long floating point number in hexadecimal floating point (HFP) or binary floating point (BFP, the IEEE standard) format**

Language	Defining characteristics	Examples
Assembler	**DC or DS instruction with data type 'D' or 'DB', respectively, and for DC an explicit value (decimal or with exponent)**	`Area_1  DC    D'75.2E2'` `In_A    DS    D`

COBOL	**No PIC clause, possibly a Value clause, and a USAGE of COMP-2 or COMPUTATIONAL-2; BFP is not supported**	`01  Area-1   comp-2`
		`          value 75.2E2.`
		`01  In-A     comp-2.`

PL/I	**DECLARE of type FLOAT DECIMAL(16) or FLOAT BINARY(53), possibly with INIT; compiler option DEFAULT(IEEE) makes all floating point BFP**	`dcl  Area_1`
		`       float decimal(16)`
		`       init(75.2E2);`
		`Dcl In_A dec float(16);`

C/C++	**declare as double (decimal or with exponent); any initial value done by assignment; compiler option FLOAT(IEEE) makes all floating point items BFP**	`double Area_1 = 75.2E2;` `double In_A;`

Data Types - Floating Point, extended

❒ **16 bytes, doubleword aligned, containing an extended floating point number in hexadecimal floating point (HFP) or binary floating point (BFP, the IEEE standard) format**

Language	Defining characteristics	Examples
Assembler	**DC or DS instruction with data type 'L' or 'LB', respectively, and for DC an explicit value (decimal or with exponent)**	`Vol_1   DC   L'18.7E16'` `In_V    DS   L`
COBOL	**Not supported**	
PL/I	**DECLARE of type FLOAT DECIMAL(33) or FLOAT BINARY(109), possibly with INIT; compiler option DEFAULT(IEEE) makes all floating point BFP**	`dcl  Vol_1 float decimal(33) init(18.7E16);` `Dcl In_V dec float(33);`
C/C++	**declare as long double ; any initial value done by assignment (decimal or with exponent); compiler option FLOAT(IEEE) makes all floating point items BFP**	`long double Vol_1 = 18.7E16;` `long double In_V;`

Data Types - Addresses

❒ **Four bytes, fullword aligned, containing a 24-bit or 31-bit memory address; an unsigned integer**

  ✗ **64-bit addressing is only discussed tangentially in this course**

❒ **Addresses are used in many different ways, including passing arguments and receiving parameters**

Assembler

- **In Assembler, you can define address constants ("adcons") of types A, V, Y, S, Q, R, and J (a suffix of "D" indicates a 64-bit address)**

  ✗ A-type adcons (A, AD) can contain
    - ➢ a positive integer
    - ➢ an address of a data item in your program
    - ➢ an address of an instruction in your program

  ✗ V-type adcons (V, VD) can contain
    - ➢ an address of an external subroutine
    - ➢ an address of an external data item

  ✗ Y-type adcons, S-type adcons, R-type adcons (R, RD), and J-type adcons (J, JD) are not discussed in this course

  ✗ Q-type adcons (Q, QD), which contain offsets, are discussed later

Data Types - Addresses, 2

COBOL

- In a COBOL program, you can define a data item as having a usage of POINTER (no picture clause, and no VALUE clause)

- You may also use the ADDRESS OF special register for linkage section items with levels 01 and 77

- The SET construct lets you populate POINTER items and ADDRESS OF values, for example (note that for SET, the direction of data movement is from the second operand to the first):

```
set mess-pointer to address of next-item
set address of table to tab-pointer
set hold-ptr to work-ptr
```

- Two pointers, two ADDRESS OF registers, or a pointer and an ADDRESS OF register can be compared, but only for equals or not equals:

```
if hold-ptr not equal next-ptr ...
```

- The special value NULL (or NULLS) is used to indicate that a particular pointer or ADDRESS OF register does not currently contain a valid address; nothing is equal to, or not equal to, NULL, it just IS NULL or it IS NOT NULL

```
if address of arg-3 is null ...
set msg-ptr to null
```

Data Types - Addresses, 3

COBOL, continued

- **Both POINTER and ADDRESS OF, when not NULL, refer to an address in memory of a data item**

- **PROCEDURE-POINTER refers to an address in memory of an executable instruction (a program entry point), as does FUNCTION-POINTER**

- **In these examples, the first operand is always a procedure-pointer**

```
set handler-ptr to other-prcd-ptr
set handler-ptr to entry pgm-name
set work-sub-ptr to entry 'MYSUB'
set work-sub-ptr to null
set work-sub-ptr to pgm-ptr
```

- **In this last case, "pgm-ptr" must contain the address of another program's entry point, as obtained from some non-COBOL program (as a parameter, say)**

- **PROCEDURE-POINTER data types are eight bytes: a four byte entry point address and a four byte work area**

- **FUNCTION-POINTER data elements are four bytes: just an entry point address**

Data Types - Addresses, 4

- Internally, COBOL uses a full word (4 bytes) of binary zeros (low-values) for the NULL value (that is, x'00000000')

- COBOL programs are not allowed to do address arithmetic (add or subtract to a POINTER or PROCEDURE-POINTER)

  - You can play games with REDEFINES, but don't

Data Types - Addresses, 5

PL/I

- **PL/I programs can define data items with type POINTER**

- **These data items can contain addresses of data, of entry points, or a NULL or SYSNULL value**

- **Some pointer arithmetic is supported (add and subtract; also POINTERADD builtin function)**

- **Comparisons of pointer data are only valid for equal or not equal**

- **Values can be placed into pointer data items in many ways, including use of builtin functions such as ADDR, POINTER, POINTERADD, and so on, as well as through READ, LOCATE and ALLOCATE statements, and assignment (as long as data types are appropriate)**

- **NULL is x'FF000000', SYSNULL is x'00000000'**

- **The Enterprise PL/I compiler has a compile-time option that can be set:**

  - ✗ DEFAULT(NULLSYS) -> NULL() builtin function should return x'00000000'

  - ✗ DEFAULT(NULL370) -> NULL() builtin function should return x'FF000000' (this is the IBM-supplied default)

Data Types - Addresses, 6

C

- **C has a pointer data type; pointers are usually defined by indicating what type of object is being pointed at by that pointer, for example**

```
float * sub_ptr;
```

  - ✗ defines a data item, "sub_ptr", that is a pointer to short floating point data (any short floating point data item)

- **However, if you define a pointer of type void, that pointer can point to any type of data item**

```
void * vdb_ptr;
```

  - ✗ defines data item, "vdb_ptr", that can point to any data item

- **A pointer is given a value through an assignment statement, for example:**

```
sub_ptr = &total;
```

  - ✗ "sub_ptr" now contains the address of "total"; "total" must have been defined as type float

  - ✗ The "&" is the "address of" operator

  - ✗ A pointer may be assigned to another pointer (see next page)

Data Types - Addresses, 7

C, continued

- **You can access the data to which a pointer refers using the indirection operator, *:**

```
sub_total =  *sub_ptr;
```

  ✗ puts into "sub_total" the value pointed at by "sub_ptr"

- **You can go the other way, too:**

```
*sub_ptr = sub_total;
```

  ✗ puts the value in "sub_total" into the variable pointed at by "sub_ptr"

- **Assignments are interesting:**

```
sub_ptr = another_ptr;
```

  ✗ puts the address in "another_ptr" into "sub_ptr"

```
*sub_ptr = *another_ptr;
```

  ✗ puts the value in the variable pointed at by "another_ptr" into the variable pointed at by "sub_ptr"

Data Types - Addresses, 8

C, continued

- **You can do address arithmetic on C/C++ pointers, and you can do any kind of compares**

- **A value of zero (x'00000000') is the NULL pointer, and, as with COBOL and PL/I, in C a value of NULL in a pointer indicates the pointer is not currently valid**

Other Data Types

- ❐ **COBOL and PL/I have the ability to describe edited fields using PICTURE clauses, specifying how data should be formatted**

  - ♦ **It's best to either format the data first then send the result as a character string, or to pass the raw data as a non-edited inherent data type and have it edited in the called program**

  - ♦ **In other words, do not try to pass data items with edit pictures in them as arguments to an external program (although it can be done in a few cases)**

- ❐ **LE services use a variety of data types, most of which we've already discussed**

  - ♦ **For C programmers, these data types are included in the leawi.h header file, and those are freely available for use anywhere in a C program**

    - ✗ Probably best to use them just for LE services, though, to maximize the portability of your code

      - ➢ In our examples, we use LE data types when we demonstrate using LE services

      - ➢ Otherwise, we use C data types, even to the point of creating our own structures, instead of LE defined data types, when our examples do not involve using LE services

Other Data Types - Bit Strings

❒ Although all computer usable data is simply a string of bits, we normally store data in the patterns discussed up to this point

❒ The various languages we are working with have varying degrees of ability to work with bit strings

- Assembler - you can define data to be type B and specify bit offsets and bit lengths; instructions are available to set on, set off, and test one or more bits in storage or in a register

- COBOL - currently has no inherent bit string data type support, although the ability to define hexadecimal literals provides some bit-related capability

- PL/I - can define data of type BIT string (both fixed length and variable length), and there are builtin functions to do bit manipulation

- C - can assign names to bits in a byte, and some functions can work with bit strings

❒ From our perspective, for passing and receiving data, we recommend you pass character strings and let the invoked program interpret the bits, rather than trying to pass bit string data

Other Data Types - Null Terminated Strings

- **Among the languages under discussion here, the null-terminated string is peculiar to C (and C++)**

- **In these languages, a character string is an array of one byte characters of arbitrary length**

  - **The data is terminated by the appearance of a null character (x'00') in the string, as opposed to some predetermined length**

    - For example, defining a field as **char[4] = "Ver2"** will generate 4 bytes and initialize the string to **Ver2**, with no terminating null; but **char[5]="One"** will reserve 5 bytes and initialize the string to **Onex'00??'** (characters One, a null, and one indeterminate byte)

- **The implication is that when C/C++ and some other language pass character strings between them, the authors of the program must agree in advance what type of strings will be used**

  - **For traditional character string, C/C++ needs to define an array of the expected size and use precise memcpy type functions to ensure padding to the specified length is done on the right with blanks (spaces), and to ensure that truncation occurs at the specified length**

  - **For null-terminated strings, non-C programs must append a null or remove a trailing null or scan the string for a null, depending on the situation**

    - The mapping between string types is not difficult, either way, just some extra care that must be taken

Defining Null Terminated Strings

❒ **You can define null-terminated strings in any language, and you can convert between fixed length strings and null-terminated strings in any language**

♦ **In Assembler, define a DS of type C, followed by a DC x'00':**

```
N_string    ds    0CL12
            dc    CL11'Here we are'
            dc    x'00'
```

♦ **However, you can also code the same thing this way:**

```
N_string    ds    CL11'Here we are',x'00'
```

♦ **And John Ehrman of IBM suggests a two-step approach:**

✗ Early in your code set up a SETC to define a null byte:

```
&N          SetC    (BYTE 0)
```

✗ Wherever you want to have a null terminated string, append this character in the value part:

```
N_string     dc     c'Here we are&N'
```

✗ This lets the length attribute include the null character automatically

Defining Null Terminated Strings, 2

- In COBOL, define an item with pic x's then give a value with a z-type literal:

```
01  N-string pic x(12) value z'Here we are'.
```

- In Enterprise PL/I, you can declare a string as type VARYINGZ; one more byte of storage is allocated than the specified length:

```
dcl N_string  char(11) varyingz  init('Here we are');
```

- In C, define an item with type char[nn] and it is implicitly null-terminated:

```
char N_string [12] = "Here we are";
```

Working With Null Terminated Strings

❒ **There are two essential activities here**

- **Given a traditional character string, convert this to a null terminated string**

  - ✗ In place, or copy to a work area; must be room for null character in addition to string
  - ✗ Find displacement of last blank (hint: often best to reverse the string and find the first non-blank in the reversed string)
  - ✗ Replace the last blank with a null character (x'00')

- **Given a null terminated string, convert this to a traditional character string**

  - ✗ Scan string to find null, then replace the null with a blank (x'40')
  - ✗ Alternatively, copy to target up to (but not including) the null; pad rest of target with blanks

❒ **We examine how to do this in each of our covered languages**

- **Note that these code samples represent one way to accomplish these tasks, and they have been tested, but there are certainly many ways to accomplish these tasks**

Working With Null Terminated Strings - Assembler

❒ Assume character string in 'work_string', defined as CL30, and need to <u>build a null-terminated string</u> in 'out_string', defined as CL31; also 'back_string' is defined as CL30:

```
*  populate target string
            mvc   out_string(30),work_string
            mvi   out_string+30,x'00'
            la    1,back_string+30
            la    3,back_string
            la    4,out_string+30
*  reverse string into back_string
            mvcin back_string,work_string+29
*  find first non_blank
            trt   back_string,nonblank_table
*  calculate address and move null into out_string
            sr    1,3
            sr    4,1
            mvi   0(4),x'00'
.
.
.
nonblank_table  dc  256x'01'
           org  nonblank_table+c' '
           dc   x'00'
           org
.
.
.
```

Working With Null Terminated Strings - Assembler, 2

❒ Now, assume 'out_string' is defined as CL31, it contains a null terminated string, which we are to convert to a traditional string into 'work_string'

```
         mvc   work_string,spaces
         xr    0,0
         la    2,work_string
         la    3,out_string
repeat   ds    0h
         mvst  2,3
         bc    1,repeat
         mvi   0(2),c' '
```

❒ Assembler programmers should be aware of the C-Assist instructions:

- CLST - Compare Logical STring; lets you compare two null terminated strings
- CUSE - Compare Until Substring Equal; searches two null terminated strings looking for matching substrings of a specified length
- MVST - MoVe STring; copies a null terminated string, stopping after moving the null
- SRST - SeaRch STring; search a string looking for the first occurrence of a character

Working With Null Terminated Strings - COBOL

❒ Assume character string in 'work-string', defined as pic x(30), and need to build a null-terminated string in 'out-string', defined as pic x(31); also 'back-string' is defined as pic x(30) and space-ctr as pic s9(4) binary:

```
move 0 to space-ctr
move spaces to out_string(1:30)
move function reverse(work-string)
      to back-string
inspect back-string tallying space-ctr
        for leading spaces
move work-string (1: 30 - space-ctr),
     to out-string(1: 30 - space-ctr)
move x'00' to out-string(31 - space-ctr:1)
```

❒ Now, assume 'out-string' is defined as pic x(31), it contains a null terminated string, which we are to convert to a traditional string into 'work-string'

```
move spaces to work-string
string out-string delimited by x'00'
         into work-string
```

Working With Null Terminated Strings - PL/I

- Assume character string in 'work_string', defined as char(30), and need to <u>build a null-terminated string</u> in 'out_string', defined as char(30) varyingz:

  - To find the last space, we work back from the end

```
dcl x fixed bin(15)    init(0);
dcl lb_found  bit(1)   init('0'B);
.
.
.
out_string = ' ';
lb_found = '0'B;
do x = 30 to 1 by -1 until (lb_found);
      if substr(work_string, x, 1) ¬= ' '
      then lb_found = '1'B;
end;

substr(out_string,1, x + 1) =
           substr(work_string, 1, x) || '00'x;
```

- Now, assume 'out_string' is defined as char(30) varyingz, it contains a null terminated string, which we are to <u>convert to a traditional string</u> into 'work_string':

```
    work_string = substr(out_string,1);
```

Working With Null Terminated Strings - C

❒ Assume a standard character string in 'work_string', defined as char[30], and need to build a null-terminated string in 'out_string', defined as char[31]:

```
short int i;
short int j;
.
.
.
for (i=0;i<30;i++) out_string[i] = work_string[i];
for (i=29;i>0;i--)
      {
       if (out_string[i] != ' ')
            {
              out_string[i+1] = '\0';
              break;
            }
      }
if (i==0) out_string[0] = '\0';
```

❒ Now, assume 'out_string' is defined as char[31], it contains a null terminated string, which we are to convert to a traditional string into 'work_string', which is defined as char [30]

```
for (j=0;j<30;j++) work_string[j] = ' ';
i = strlen(out_string);
for (j=0;j<i;j++) work_string[j] = out_string[j];
```

Rules For Names

- **Just as a convenience, we've summarized the rules for creating user names for data items in the languages we are examining**

Assembler

- ✗ 1-63 alphanumeric, national (@, #, $) and underscore characters
- ✗ first must not be numeric
- ✗ unique within a source program
- ✗ case-insensitive

COBOL

- ✗ 1-30 alphanumeric and hyphen (dash) characters; as of Enterprise COBOL 4.2, the underscore is also allowed
- ✗ first and last must not be hyphen; first must not be underscore
- ✗ must contain at least one alphabetic character
- ✗ unique within a data structure
- ✗ may not be a reserved word
- ✗ case-insensitive

PL/I

- ✗ 1-31 alphanumeric, extralingual, and underscore characters; Enterprise PL/I: allows up to 100 characters (depending on a compiler option); extralingual characters default to $, #, @, but you can choose your own based on a compiler option
- ✗ first must be alphabetic, extralingual, or underscore
- ✗ unique within a data structure
- ✗ case-insensitive

Rules For Names, 2

C

- ✗ unlimited alphanumeric and underscore characters, but must be unique within the first 255 characters
- ✗ first must not be numeric
- ✗ unique within scope
- ✗ case-sensitive

Note

- ♦ **If you give a subroutine a name that begins with 'IBM', 'PLI' or 'CEE', C will change the name by converting the third character to '$'**
  - ✗ C wants to ensure that needed support routines can't be used for user routine names
- ♦ **Note that this is regardless of the language the subroutine is coded in**
  - ✗ Just an alert

❒ **In MVS and OS/390, external names (program names, member names, sometimes ddnames) have historically been limited to 8 characters (7 in PL/I), which must also be only upper case**

- ♦ **z/OS supports external names that are up to 1024 characters long (32767 characters in z/OS 1.3 and later), and that are case sensitive; details later**

Computer Exercise #2: Defining Elementary Data Items

In the libraries you created as part of the previous lab you will find a variety of mainline and subroutine source modules.

It is expected that you will want to work with mainline code in only one language (although you are welcome to work with mainlines in as many languages as you choose). We do expect everyone to work with subroutines in all languages for which you have a compiler.

To this end, we have provided skeleton code, comments with lots of clues, and some lab assist programs, macros, copy books, and so on.

For this lab, you should define some data elements in the mainline program for the language of your choice, from the list:

M52MNA1	Assembler
M52MNC1	COBOL
M52MNP1	PL/I
M52MND1	C

Define these elements (please use these names, attributes, and initial values; use language appropriate punctuation and syntax, of course; follow the instructions in the code for Exercise 2; in COBOL programs: replace all underscores ('_') below with dashes ('-') in the program):

Name	Attributes	Initial value
char_5	5 byte standard characters string	'Taste'
char_5n	5 byte null terminated string (6 bytes total)	'Paste'x'00'
pack_52	packed decimal; 7 digits, 2 to right of decimal	35.33
bin_half	binary halfword	1234
bin_full	binary fullword	123456789
flt_short	short floating point	8.0e1
flt_long	long floating point	5.0e1

Also note for COBOL programmers: depending on which version of the COBOL compiler you are using, you may need to change the string in the first line that says **test(nohook)** to be **test(sym,none)** or maybe just **test**.

Computer Exercise #2: Defining Elementary Data Items, 2

We have provided several JCL members in your <hlq>.TR.CNTL library. For right now, these members might be of interest:

ASMSUBC	- assemble and bind an assembler program
COBSUBC	- compile and bind a COBOL program
PLISUBC	- compile and bind a PL/I program
CSUBC	- compile and bind a C program

In each case, the xxxSUBC members have a line:

```
//          SET  O=
```

To use this JCL to Assemble or compile a program, fill in the program name after the O= (with no intervening spaces), for example:

```
//          SET  O=M52MNA1
```

if you are Assembling and binding the Assembler program.

Each job has a jobname that is your high level qualifier with a '1' appended; you may wish to change the last letter in each jobname.

So, in the appropriate JCL member, set the O= value to be the name of the mainline program you modified.

Finally, to test the syntax of your code, run the appropriate job to Assemble / compile and bind the mainline program you modified. Check your results and fix any errors.

> **Exercise Stretch**: Do the above for one or more addtional mainline programs.