Cross Program Communication in z/OS

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

Object Code
- Modules
- Module Translations
- Sections
- Object Modules
- Object Modules: XOBJ
- Generated Object Modules

Passing Arguments and Receiving Returned Values
- How Arguments Are Passed - Styles and Options
- How Arguments Are Passed - Assembler, COBOL, PL/I, C
- How Arguments Are Passed - Lessons

Receiving Parameters and Setting Return Values
- Mainlines and Subroutines
- Subroutine declarations
- Declaring Parameters
- Parameters - Assembler, COBOL, PL/I, C
- Computer Exercise: Assemble / compile, bind subroutines 259

The Program Binder
- Compiles and Binds
- Assemble / Compile and Bind Data Flow
- An Example
- Program Binder PARM Options
- Program Binder Control Statements: ENTRY, NAME
- A Load Module
- Program Binder Control Statements: INCLUDE, LIBRARY, REPLACE
- How The Program Binder Works
- Basic Maintenance Using the Program Binder
- Computer Exercise: Program Binder and Maintenance 294

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 (LE-enabled)

_____ C

_____ COBOL

_____ PL/I

Note: This course can focus on one, two, three, or four programming languages. The selected mix and the style of the instructor will affect the actual number of days the course will last.

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 in these notes**

  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, we will not show standard save area linkage code unless it is required to demonstrate some aspect of the example**

We will not show the LE Assembler macros, 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 will not necessarily show the target of branch instructions, 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, we will not show any code not necessary for understanding of an example

  We assume familiarity with PL/I program structure

  We will not generally show declarations for builtin functions nor LE service routines

- We will not necessarily show the target of "call" 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:

```
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, we will not show any code not necessary for understanding of an example**

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 will not necessarily show the target of function references, 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.PDSE	for program objects