Introduction to Application Programming (z/OS)

Introduction to Application Programming (z/OS) - Course Objectives

On successful completion of this class, the student should be able to:

1. Describe the major issues in program design

2. Describe inputs and outputs for a program, down to the field level

3. Design program logic for basic programs

4. Describe the steps necessary to complete the process to code, compile, link, and test a program

5. Describe these fundamental data types of IBM mainframe machines: character, packed decimal, binary

6. Convert numbers between binary and decimal and hexadecimal

7. Perform basic arithemtic with binary and hexadecimal numbers.

Introduction to Application Programming (z/OS) - Topical Outline

Day One

Introduction to Application Programming (z/OS) - Topical Outline, p.3

Day Two, continued

Behind The Scenes - Software
- Virtual Storage Concepts
- z/OS Architecture
- Batch Application Environments
- Online Application Environments

- What's Next?

Section Preview

❒ **Introduction To Application Programming**

- **Computer Applications**
- **The Application Programmer Job**
- **Platforms**
- **Program Functions**
- **Program Design**
- **The Output - Describing What We Want**
- **The Input - Describing What We've Got**
- **Data**
- **Data Organizing**
- **Pseudo-Descriptions**
- **Describing Data (Excercise)**

Computer Applications

Work run on a computer to accomplish a business function, or to support a business function

Financials

Accounting

Planning

Research and Development

Product Design

Manufacturing / Production

Marketing / Sales

Customer Services / Support

Application Utilities (ancillary services)

- ✗ **Data Entry / Validation / Storage**
- ✗ **Data Modification**
- ✗ **Data Backup / Recovery**

The Application Programmer Job

- The work of an application programmer includes ...

  Analyzing user requirements

  Refining requirements

  Researching data needs

  Designing files (data layout)

  Designing application flow

  Designing programs

  Coding programs

  Testing programs

  Documenting programs

  Training users on how to use programs

  Maintaining programs

  And, of course, the usual helping of politics, meetings, communication, ego-salving, etc., etc., ...

- In this course, we focus just on creating new programs: part of the technical aspects of the job

Platforms

Applications, in a computer system environment, are built on existing, available bases, or <u>platforms</u>

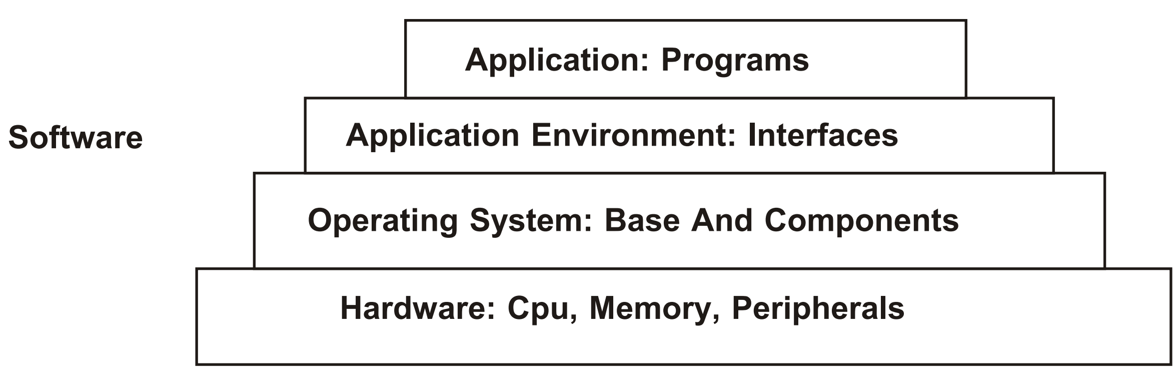
- ❒ **An application is made up of component pieces, called <u>programs</u>**

- ❒ **In olden times (ten or twenty years ago), application programs rested directly on the Operating System**

- ❒ **In modern times, applications are built upon an Application Environment**

    - ✗ **The Application Environment takes care of using the Operating System services to get work done**

- ❒ **Finally, the Operating System rests on the underlying hardware platform (IBM mainframes in this seminar)**



Programming Languages

- There are hundreds of programming languages to choose from in the world, but in the IBM mainframe application programming niche, programs are generally written in these languages

    COBOL (COmmon Business Oriented Language)

    PL/I (Programming Language / I)

    C

    Assembler Language (sometimes called ALC or BAL)

- We will be giving examples of programs written in all of these languages, but we will focus most on COBOL since that is far and away the most widely used language in this environment

- A programming language has a limited vocabulary and syntax that can be combined in a great many ways to produce the desired results

    A computer cannot directly execute the instructions written in a programming language: <u>a computer can only execute machine instructions</u>

    So computer programs need to be translated into a set of machine instructions (we say the program needs to be <u>compiled</u>)

Program Functions

❒ **Every business application program performs one (or more) of these functions:**

Read input data from tape, disk, a terminal, or some other external device

Locate related data, on tape or disk or other external medium

Process data

✗ Display data on screen (real time, interactive) or hardcopy (reports, invoices, bills, statements)

✗ Add new data to data already an external medium

✗ Update (change, modify) existing data on an external medium

✗ Delete data from an external medium

❒ **Note that data are generally organized as <u>records</u> in <u>files</u> (more later)**

❒ **A collection of data and the programs to process the data is called a <u>computer application</u>**

Program Design

❐ **Before we can write a program, we first have to design it, which traditionally has three basic parts:**

> **We need to know what the user wants (what is the <u>output</u>, the result, of the program)**
>
> **We need to know what's already available (where is the <u>input</u>, the data we need in order to produce the output)**
>
> **We need to figure out how to get from the starting situation to the desired end point (what <u>process</u> must we go through to get where we want to go)**

❐ **Some people make an analogy between a program and a recipe**

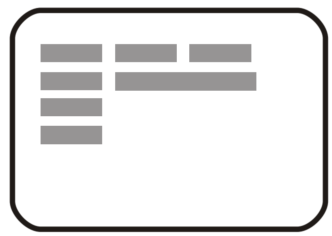
> **A recipe describes the dish (output), the ingredients (input), and the process (directions) to produce the dish**
>
> **A program does the same for data**
>
> **The analogy works pretty well, but don't get too carried away, because we shall see some places where the analogy breaks down**

The Output - Describing What We Want

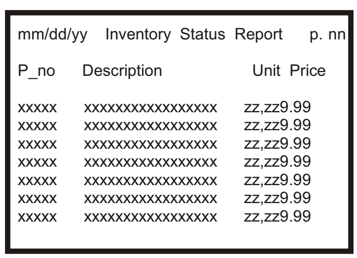
❐ **So, how do you describe output?**

It depends

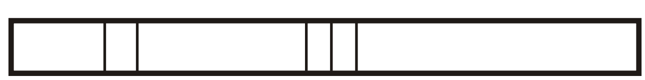
For a data display, we can draw a map of a terminal screen and describe where each piece of data goes, how it should be formatted, and so on ...



For a report, we draw a sample page layout



For an update to files on tape or disk, we describe the data layouts and how these layouts are impacted by our program



The Input - Describing What We've Got

❒ **Similarly, we have data available to us on tape or disk and we have the capability to accept data from a terminal**

❒ **How do we describe this data?**

Input data on tape and disk are described in the same ways as output data going to tape or disk: we describe the record layouts

Input data from a terminal is described as individual pieces: <u>fields</u> (more detail in just a minute)

❒ **Let us try to get this more concrete ...**

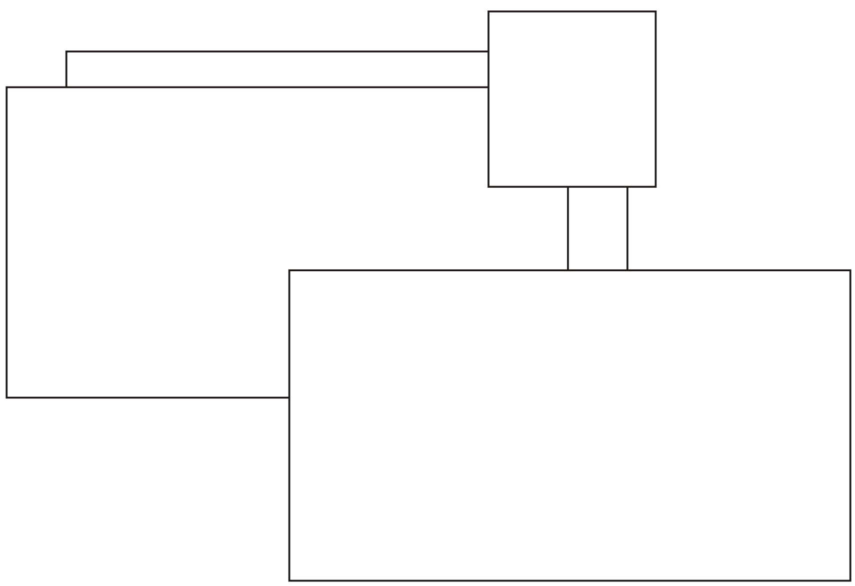
Data

❒ **In the non-computer environment, data is most often on paper**

Hand-written or typed sheets of paper / forms

Informal jottings on a scrap of paper

Could be photographs, pictures, or audio or video tapes!



❒ **Really organized people even group data into folders**

Maybe even use credenzas, filing cabinets, storage bins, and so on

Data Organizing

❒ **Computers work best with data that is structured and organized**

And in machine-readable form

❒ **The basic structuring of data for use in a computer can be thought of as making lists**

The data stored on a piece of paper that represents an item in inventory, for example, will be one entry in the list:

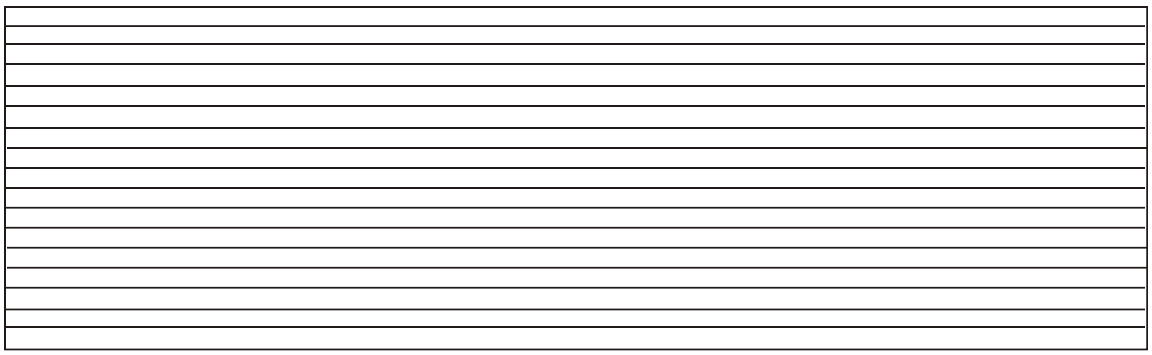
Inventory Item

Part Number: TUB-345/X
Quantity on hand: 50
Unit Price: 13.225
Description: Pink Tubing
Date Last Order: 06/05/20xx
Quantity Last Ordered: 30
Last Price: 12.285
Supplier: BTRX-88-01
. . .

```
TUB-345/X00050013225Pink Tubing     06/05/20xx . . .
```

Records and Files

❒ **The collection of all our inventory items in this list, then, might look like this:**



Each "strip" or "line" represents one of our items in inventory

✗ We call each of these entities a <u>record</u>

✗ We call the entire list, the list as a whole, a <u>file</u>

✗ In the IBM mainframe environment, we use the term <u>data set</u> the same as <u>file</u>

❒ **In the computer world, files are stored on magnetic disk or tape, or on optical media such as CDs (compact disks)**

Files are given names, and the file names themselves are stored in catalogs, so the system can locate the files when our programs are ready to process the records

Fields

❒ **Records are composed of fields, the individual pieces of information that make up the record**

For example, in an inventory data set we have records that have these fields:

- ✗ Part number
- ✗ Quantity on hand
- ✗ Unit price
- ✗ and so on

❒ **Each field describes some characteristic of the object that a record describes**

Field Size

❒ **To specify the contents of a record, we need to know what fields the record contains, and how many characters each field contains**

❒ **How big is each field?**

"It depends"

An item description field, for example, only needs to be five characters long if the description is "Fates"

But if the description is "Slightly used, highly burnished, plaid zinc and copper noodles", we will need considerably more space

❒ **While computers can handle varying size fields, programming is simpler and performance is better if we work with fixed length fields**

So we normally select the maximum size we want to allow for a field and use that

This means sometimes you have to make some compromises, such as abbreviating the data in a field

✗ For example, if we chose a field length of 30 for our item description, we would have to enter the second item above using abbreviations, something like this:

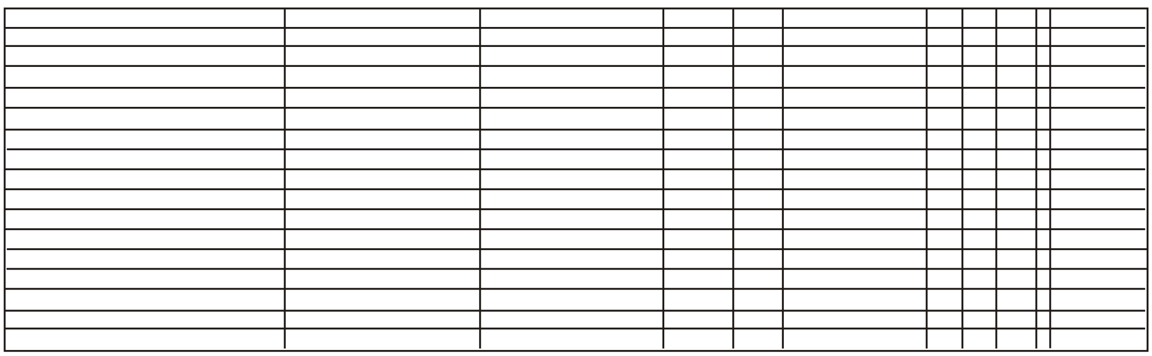
"Used,brnshd,pld,Zn+Cu noodles"

Records and Fields

❒ **So a record is made up of fields**

And each record in a file is typically composed of the same fields - that is, each record usually has the same general structure

Visually, it might look something like this:



Describing Fields

❒ **To describe a record, you list the fields that make up the record, in the order the fields appear in the record, specifying at least**

Field name

Rules for names in a minute

Field location

Where in the record is the field located? (starting location)

Field size

Length, in characters or bytes

- ✗ The sum of the field sizes is the size of the record

Field Names

❒ **Everything we do in computers has to be precise**

Including assigning names

<u>In a program:</u>

❒ **Every <u>file</u> has to have a <u>name</u> <u>unique within the program</u>**

Each record in the file has to be described as a structure composed of fields

❒ **Every <u>record</u> <u>or</u> <u>data structure</u> has to have a <u>name</u> <u>unique relative to a file description</u>**

❒ **Every <u>field</u> has to have a <u>name</u> <u>unique within</u> a <u>record or data structure</u> (there is one exception we discuss later)**

❒ **Also, there may be fields that exist independent of any record structure (an item used to hold a calculation, for example)**

Each <u>independent data item</u> must have a <u>name</u> <u>unique within the program</u>

Field Names, 2

❒ **The rules for names also vary depending on the programming language you are coding in**

Maximum length of a name varies from 8 characters to over 60 characters (COBOL: 30)

Characters allowed in a name are usually alphabetic (A-Z) or numeric (0-9) (for example: Description**,** AddressLine2**)**

Most languages allow certain special characters to appear in names

✗ But spaces (blanks) are never allowed in a name in a mainframe environment

➢ So most languages designate a "break character" that can separate the parts that would normally be done by a space

➤ In COBOL, a dash, so, for example: **Unit-Price**

➤ In PL/I, Assembler, and C, an underscore: **Unit_Price** (note that in the latest COBOL compilers, you may use an underscore in a name)

Capitalization of names matters in C, but not in most other languages

✗ So **PartNumber** and **PARTNUMBER** would be considered to be the same field name, except in C

Record Structures

- ❐ **When a number of fields are related, such as the fields that make up a record, we generally group them together under a record name**

  For example, a personnel file might contain employee records, and these records be might be made up of fields such as these

Sample Fields and Data Structure Description

```
Employee-record.
    Employee-number (6 characters)
    Last-name (15 characters)
    First-name (12 characters)
    Middle-init (1 character)
    Hire-date.
        Hire-year (4 characters)
        Hire-month (2 characters)
        Hire-day (2 characters)
    Salary-type (2 characters)
    Salary (9 characters, all numeric digits,
            including 2 digits for cents)
    AddressLine1 (40 characters)
    AddressLine2 (40 characters)
    City (35 characters)
    State (2 characters)
    Country (35 characters)
    MailCode (15 characters)
```

Notice names, structures, sub-structures, and how items are described

Principle of Completeness

❒ **Suppose you are describing records in a file**

And that the records contain fifty different fields

But for this program you are only referencing three fields

✗ You must still account for every character in the record, because data are stored on tape or disk or printers in complete records

✗ Your program is always passed a complete record on input

✗ Your program always writes a complete record on output

We call this the principle of completeness

Naming Unreferenced Fields

❒ **As we just saw, even if your program is not referencing every field you still need to account for every field - the space it takes up**

❒ **But if a field is unreferenced in your program you can give it a "don't care" kind of name, like "Unused"**

But since field names must be unique within a structure, you might have to use names like "Unused01", "Unused02", and so on

❒ **Except that COBOL allows you to use the reserved word FILLER for fields that won't be referenced in a program**

And with the latest version of COBOL you can even omit the field name for unreferenced fields (although you must still account for all the space in your record)

❒ **Also, Assembler lets you reserve space for fields without assigning a name of any kind**

Other Data Structures

- ❒ **Normally in a program you need to <u>define</u> (or <u>declare</u> or <u>describe</u>) a structure for every record in every file the program is processing**

- ❒ **And, you may use structures in your program that are not related to particular files**

  Intermediate work areas, for examples, or tables kept in memory while you work

  These, too, if used in your program, must be described in your program

- ❒ **Defining (describing) data gives the compiler information it needs when it is translating your source program into machine instructions**

- ❒ **Every independent item and every data structure must be declared**

Declaring Files

❐ **In addition to declaring all the data items used in your program, you must also define the files**

This helps tie together records in your program to files outside of your program

❐ **Declaring a file typically includes**

Assigning a file name for use inside your program

✗ Remember, file names must be unique within a program

Specifying the characteristics of the file

✗ The level of detail needed depends on the programming language

Identifying which records come from / go to which file

✗ So the compiler can make sure data flows between buffers and work areas properly

Pseudo-Descriptions

- ❒ **In the data design process, you don't have to follow particular language rules for names for files, records, and data items**

  Just rough things out

  You might call this a "pseudo-description" (a "sort of" description)

  Just so you can focus on design and not worry about syntax

- ❒ **However, if you know you will be coding in a particular language it is probably a good idea to follow the rules for names for that language**

  This way, you don't need to change much when moving from design to coding

Exercise: Describing Data

For our first exercise, you have been assigned the task of creating a report that lists every item in our inventory file.

The instructor is to play the role of user, the warehouse manager. You may ask any question you think relevant to the task at hand. Remember that the instructor is playing a role, and will answer questions in the way the user might answer them.

When you are done, write down, on this page and the next, the contents of the report (along with a sketch), and the description of where the output data comes from.