COMSATS University
Islamabad
(Lahore Campus)

CSC103-Programming Fundamentals

MS. MAHWISH WAQAS

Mawish.waqas@cuilahore.edu.pk

Chapter 1:
An Overview of Computers and Programming Languages

Resources

Course Website

- https://sites.google.com/cuilahore.edu.pk/csc103-pf/csc103-pf
- Contains lectures, labs, books/notes and completed assessments (with solutions)

Objectives

In this chapter, you will:

- Learn about different types of computers
- Explore the hardware and software components of a computer system
- Learn about the language of a computer
- Learn about the history of C++ Programming Language
- Understand the process of a C++ Program execution

Objectives (cont'd.)

- Discover what a compiler is and what it does
- Examine a C++ program
- Explore how a C++ program is processed
- Learn what an algorithm is and explore problem-solving techniques
- Become aware of structured design and object-oriented design programming methodologies
- Become aware of Standard C++, ANSI/ISO Standard C++, and C++11

Introduction

Without software, the computer is useless

Software is developed with programming languages

- C++ is a programming language

C++ suited for a wide variety of programming tasks

The Language of a Computer

Analog signals: continuous wave forms

Digital signals: sequences of 0s and 1s

Machine language: language of a computer; a sequence of 0s and 1s

Binary digit (bit): the digit 0 or 1

Binary code (binary number): a sequence of 0s and 1s

The Evolution of Programming Languages

Early computers were programmed in machine language

To calculate `wages = rate * hours` in machine language:

```
100100 010001     //Load

100110 010010     //Multiply

100010 010011     //Store
```

The Evolution of Programming Languages (cont'd.)

Assembly language instructions are mnemonic

Assembler: translates a program written in assembly language into machine language

Using assembly language instructions, `wages = rate • hours` can be written as:

```
LOAD  rate
MULT      hour
STOR      wages
```

The Evolution of Programming Languages (cont'd.)

High-level languages include Basic, FORTRAN, COBOL, Pascal, C, C++, C#, and Java, Python

Compiler: translates a program written in a high-level language into machine language

Two Major Programming Pardiagms

Structured programming

Object oriented programming

Structured programming

Dividing a problem into smaller subproblems is called *structured design*.

- Each subproblem is then analyzed, and a solution is obtained to solve the subproblem.
- The solutions to all of the subproblems are then combined to solve the overall problem.
- The unit of program is a *function*.

This process of implementing a *structured design* is called **structured programming**.

- The structured-design approach is also known as top-down design, stepwise refinement, and modular programming.

Object Oriented Programming

Object-oriented design (OOD) is a widely used programming methodology.

- In OOD, the first step in the problem-solving process is to identify the components called **objects**, which form the basis of the solution, and to determine how these objects interact with one another.
- The unit of program is an *object*.

Each object consists of data and operations on that data, e.g. a book has a price (data), and it can be purchased (operation on data).

- In OOD, the final program is a collection of interacting objects.
- A programming language that implements OOD is called an object-oriented programming (OOP) language.
  - You will learn about OOP in the next semester.

Processing a C++ Program (cont'd.)

To execute a C++ program:

1. Use an editor to create a source program in C++
2. **Preprocessor directives** begin with # and are processed by the preprocessor (a separate program)
3. Use the compiler to:

- Check that the program obeys the language rules
- Translate into machine language (**object program**)
  - Source code vs object code

A sample C++ Program (Source code)

```
#include <iostream>

using namespace std;

int main()
{
  cout << "My first C++ program.“ ;
  return 0;
}
```

Sample Run:

```
My first C++ program.
```

A sample C++ Program (Source code)

```
#include <iostream>



int main()
{
    std::cout << "My first C++ program.“ ;
  return 0;
}
```

Sample Run:

```
My first C++ program.
```

Processing a C++ Program (cont'd.)

To execute a C++ program (cont'd.):

4. Linker:
   - Combines object program with other programs provided by the C++ Software Development Kit (SDK) to create an executable code
   - Library: contains prewritten code you can use
5. Loader:
   - Loads executable program into main memory

- The last step is to execute the program

Some software (called IDEs) do all these steps with a single Build or Rebuild command.

- More about this later.

Processing a C++ Program (cont’d.)

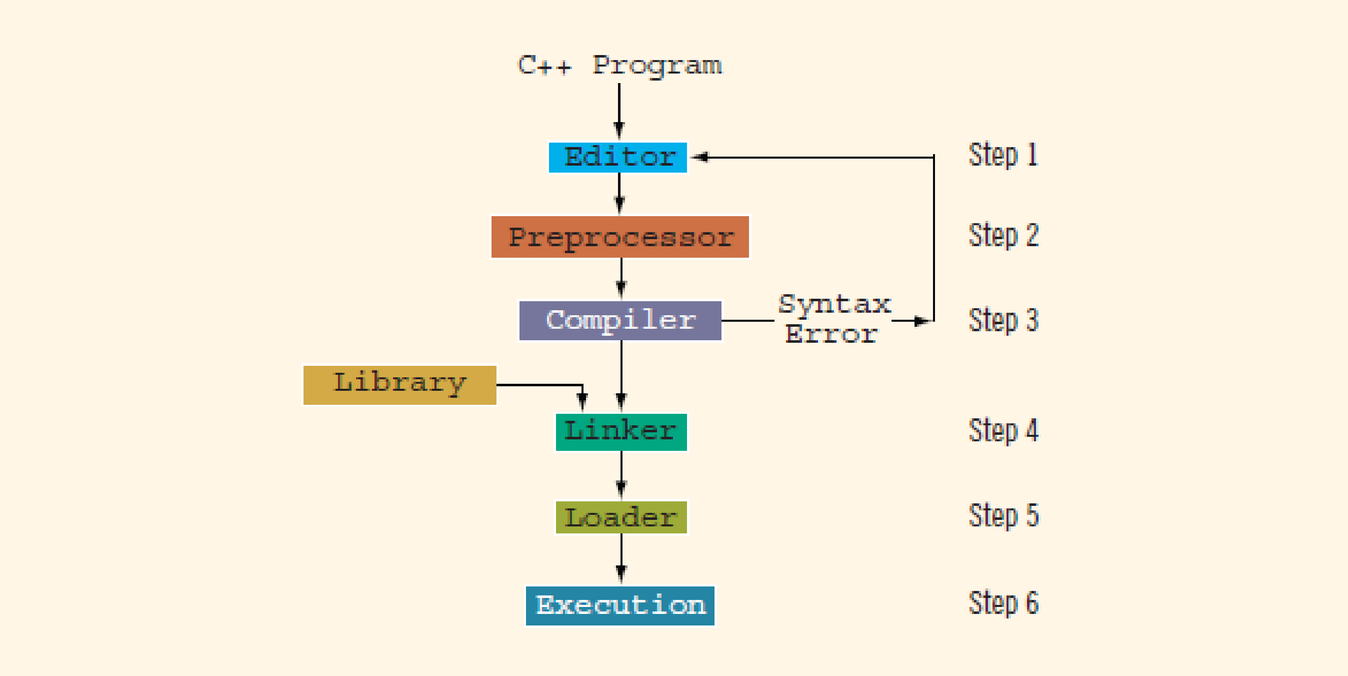

FIGURE 1-2 Processing a C++ program

Home Task

Download CodeBlocks (with gcc compiler) from:

- https://www.fosshub.com/Code-Blocks.html?dwl=codeblocks-20.03mingw-setup.exe

Compile HelloWorld.cpp using:

- Gcc HelloWorld.cpp
  - If the code is incorrect, then error messages are shown along with line numbers. Otherwise, This will generate an exe file (object code) named a.exe.
- You can give the output file name yourself with the following command:
  - gcc -o Hello.exe HelloWorld.cpp
    - This will generate Hello.exe output file.

Programming with the Problem Analysis–Coding–Execution Cycle

<u>Algorithm</u>:

- Step-by-step problem-solving process
- Solution achieved in finite amount of time

Programming is a process of problem solving

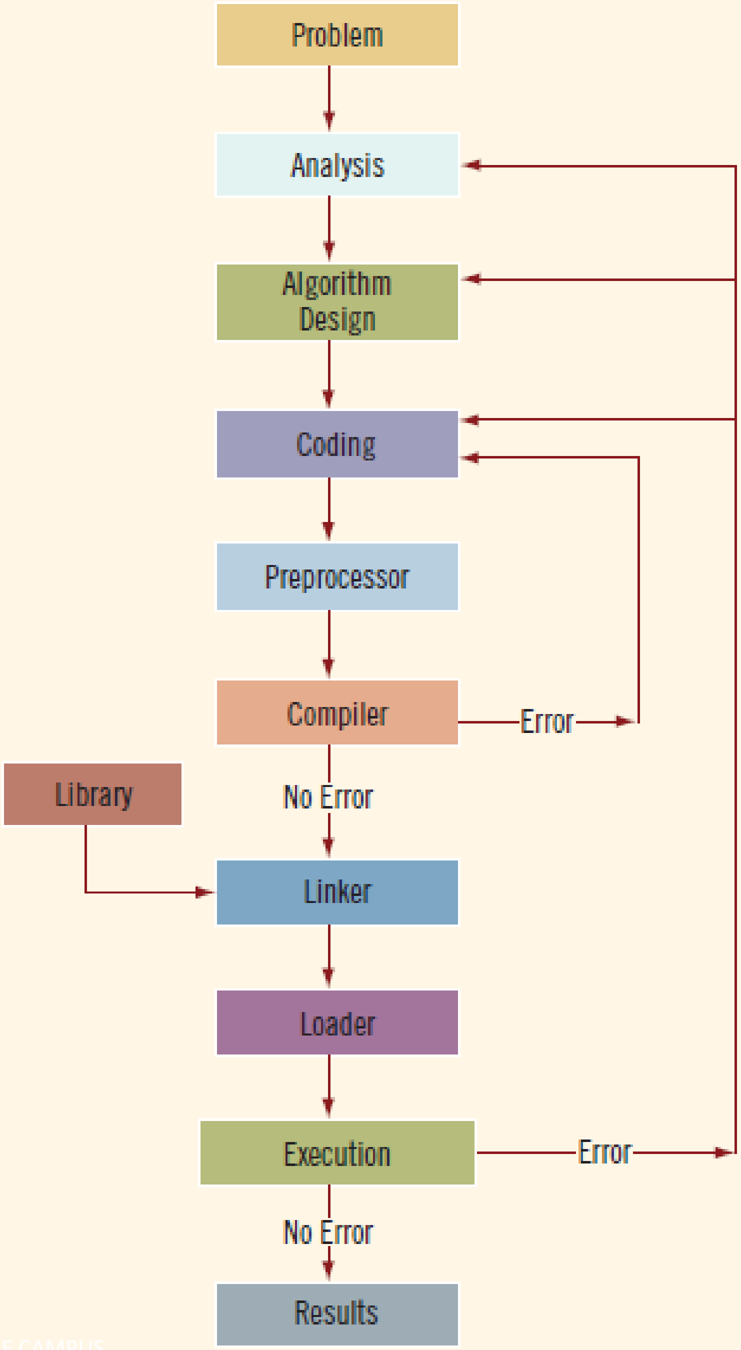

FIGURE 1-3 Problem analysis–coding–execution cycle

The Problem Analysis–Coding–Execution Cycle (cont'd.)

Step 1: Analyze the problem

- Outline the problem and its requirements
- Design steps (algorithm) to solve the problem

Step 2: Implement the algorithm

- Implement the algorithm in code
- Verify that the algorithm works

Step 3: Maintain

- Use and modify the program if the problem domain changes

The Problem Analysis–Coding–Execution Cycle (cont’d.)

Thoroughly understand the problem and all requirements

- Does program require user interaction?
- Does program manipulate data?
- What is the output?

If the problem is complex, divide it into subproblems

- Analyze and design algorithms for each subproblem

Check the correctness of algorithm

- Can test using sample data
- Some mathematical analysis might be required

The Problem Analysis–Coding–Execution Cycle (cont’d.)

Once the algorithm is designed and correctness verified

- Write the equivalent code in high-level language

Enter the program using text editor

The Problem Analysis–Coding–Execution Cycle (cont'd.)

Run code through compiler

If the compiler generates errors

- Look at code and remove errors
- Run code again through compiler

If there are no **syntax errors**

- Compiler generates equivalent machine code

Linker links machine code with system resources

The Problem Analysis–Coding–Execution Cycle (cont’d.)

Once compiled and linked, loader can place program into main memory for execution

The final step is to execute the program

Compiler guarantees that the program follows the rules (*syntax*) of the language, i.e. it identifies *syntax* errors

- Does not guarantee that the program produces correct output at execution/run time, i.e. it does not identify *logical* and *runtime* errors.
  - E.g. if a compiler cannot tell whether you have used the correct formula for quadratic roots in your program.

Example 1-1

Design an algorithm to find the perimeter and area of a rectangle

The perimeter and area of the rectangle are given by the following formulas:

```
perimeter = 2 * (length + width)

area = length * width
```

Example 1-1 (cont'd.)

Algorithm:

- Get length of the rectangle
- Get width of the rectangle
- Find the perimeter using the following equation:

```
perimeter = 2 * (length + width)
```

- Find the area using the following equation:

```
area = length * width
```

Example 1-5

Calculate each student’s grade

- 10 students in a class; each student has taken five tests; each test is worth 100 points

Design algorithms to:

- Calculate the grade for each student and class average
- Find the average test score
- Determine the grade

Data: students’ names; test scores

Example 1-5 (cont'd.)

Algorithm to determine the average test score:

- Get the five test scores
- Add the five test scores
    - Suppose `sum` stands for the sum of the test scores
- Suppose `average` stands for the average test score:
    - `average = sum / 5;`

Example 1-5 (cont'd.)

Algorithm to determine the grade:

```
 if average is greater than or equal to 90
    grade = A
otherwise
 if average is greater than or equal to 80 and less than 90
       grade = B
otherwise
 if average is greater than or equal to 70 and less than 80
       grade = C
otherwise
         if average is greater than or equal to 60 and less than 70
       grade = D
otherwise
       grade = F
```

Example 1-5 (cont'd.)

Main algorithm is as follows:

- `totalAverage` = 0;
- Repeat the following for each student:
  - Get student's name
  - Use the algorithm to find the average test score
  - Use the algorithm to find the grade
  - Update `totalAverage` by adding current student's average test score
- Determine the class average as follows:
  - `classAverage = totalAverage / 10`

ANSI/ISO Standard C++

C++ evolved from C

C++ designed by Bjarne Stroustrup at Bell Laboratories in early 1980s

- Many different C++ compilers were available

C++ programs were not always portable from one compiler to another

In mid-1998, ANSI/ISO C++ language standards were approved

Second standard called C++11 approved in 2011

Summary

Compiler: translates high-level language into machine code

Algorithm: step-by-step problem-solving process; solution in finite amount of time

Problem-solving process has three steps:

- Analyze problem and design an algorithm
- Implement the algorithm in code
- Maintain the program

Summary (cont’d.)

Structured design:

- Problem is divided into smaller subproblems
- Each subproblem is solved
- Combine solutions to all subproblems

Object-oriented design (OOD): a program is a collection of interacting objects

- Object: data and operations on those data

Reading Assignment

1. Read and understand all the examples given at the end of chapter 1 (Example 1.1 – 1.5).