Object Oriented Programming Introduction

Lecture No. 1

Lecture Contents

- Introduction/History of C/C++
- C++ Programming basics
- Pointers
- Functions

- Structures
- Classes (OOP basics, inheritance,

…) • Operator overloading

- Streams and Files
- Templates and Exceptions

Class Activity

- Write a program that generates the following output, using only one *cout statement* and *proper* spacing

1900 2135

1950 2235

2000 2335

2050 2435

History of C and C++

- **History of C**
- C is a programming language developed in the 1970's by **Dennis Ritchie (Bell Lab)** alongside the UNIX operating system
- C provides a comprehensive set of features for handling a wide variety of applications, such as systems development and scientific computation

History of C and C++

- **History of C++**
  - Early 1980s: **Bjarne Stroustrup** (Bell Lab)
  - Provides capabilities for object-oriented programming
    - **Objects:** reusable software components
    - Object-oriented programs

- **"Building block** approach" to creating programs •
  C++ programs are built from pieces called classes and functions
  - **C++ standard library:** Rich collections of existing classes and functions

Structured/OO Programming

- **Structured programming (1960s)**
  - Disciplined approach to writing programs
  - Clear, easy to test and debug, and easy to modify

- **OOP**
  - “Software reuse”
  - “Modularity”
  - “Extensible”
  - More understandable, better organized and easier to maintain than procedural programming

Basics of a Typical C++ Environment

- **C++ systems**

- Program-development environment
  - Integrated Development Environment (IDE)
- Language
- C++ Standard Library

- **C++ program names extensions**
  - .cpp (C Plus Plus)
  - .c (C)

The C++ Standard Library

C/C++ programs consist of pieces/modules called functions

- A programmer can create his own functions •

Advantage: the programmer knows exactly how it works •
Disadvantage: time consuming

- Programmers will often use the C/C++ library functions • Use these as building blocks
- Avoid re-inventing the wheel

• If a pre-made function exists, generally best to use it rather than write your own

• Library functions carefully written, efficient, and portable

Programming Style

C++ is a free-format language, which means that:

- Extra blanks (spaces) or tabs before or after identifiers/operators are ignored
- Blank lines are ignored by the compiler just like comments
- Code can be indented in any way
- There can be more than one statement on a single line

- A single statement can continue over several

lines

Programming Style (cont.)

In order to improve the readability of your program, use the following conventions:

- Start the program with a **header** that tells what the program does
- Use **meaningful variable names and Camel notation** • **Document** each variable declaration with a comment

telling what the variable is used for

• Place each **executable statement** on a **single line** • A segment of code is a sequence of executable statements that belong together

  • Use blank lines to separate different segments of code • Document each segment of code with a comment telling what the segment does.

C++ keywords

• Keywords appear in **blue** in Visual C++

• Each keyword has a predefined purpose in the language • Do not use keywords as variable and constant

names!!

- We shall cover most of the following keywords in this class:

bool, break, case, char, const, continue, do, default, double, else, extern, false, float, for, if, int, long, namespace, return, short, static, struct, switch, typedef, true, unsigned, void, while

Structure of a C++ Program

A C++ program is a collection of definitions and declarations:

- data type definitions
- global data declarations
- function definitions (subroutines)
- class definitions
- a special function called
  - **main()** (where the action starts)

General form of a C++ program

```
// Program description
#include directives
global declarations
int main()
{
 constant declarations
 variable declarations
 executable statements
 return 0;
}
```

Making Software

Problem

Writing code in “C++” language means, we must have to follow a certain rules of “C++” language.

This is called syntax of “C++” language.

Making Software

Compiler

“Turbo C” in DOS

“gcc”or “g++” in Linux

Problem

DOS/Windows.

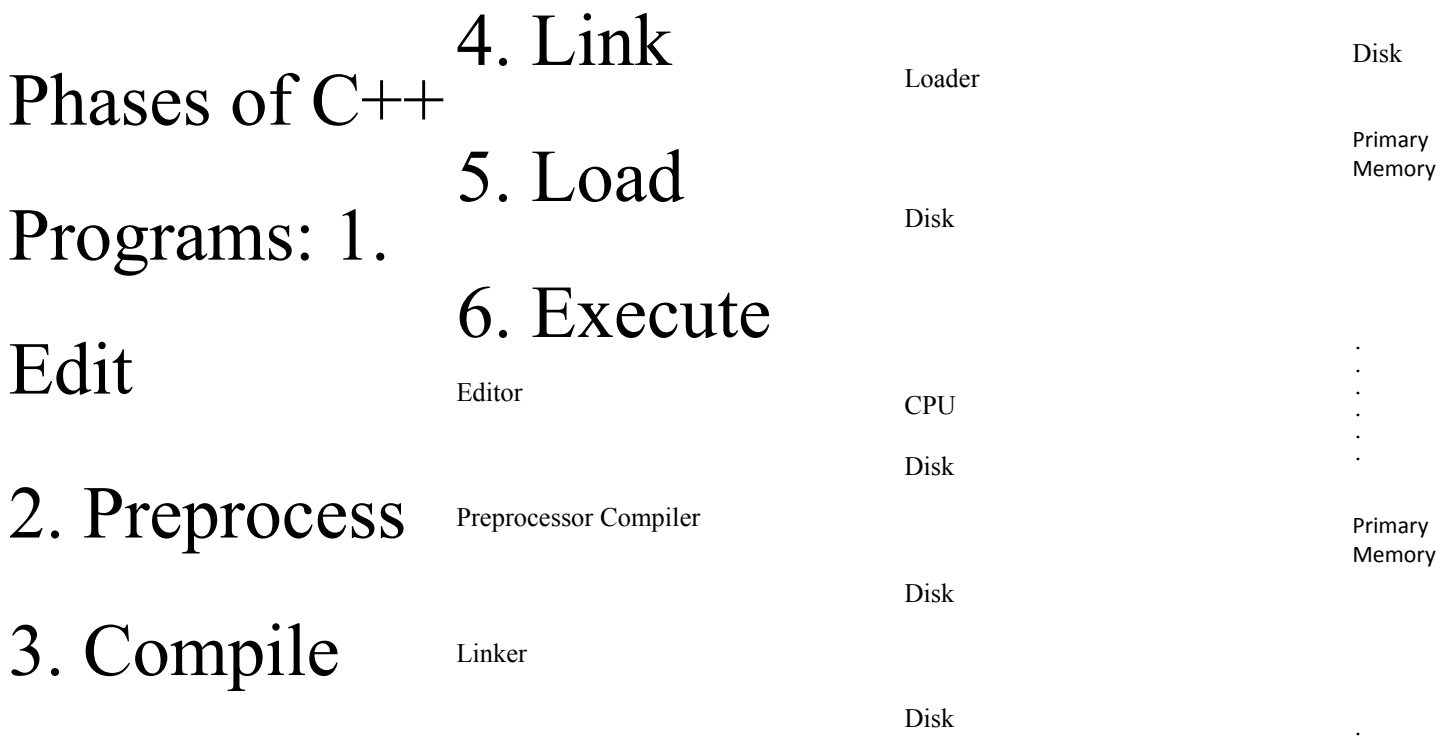
“Visual C++” in

windows “Borland” in

Writing code in “C++” language means, we must have to follow a certain rules of “C++” language.

This is called syntax of “C++” language.

Basics of a Typical C++ Environment



.
.
.
.
.
Program is created in the editor and stored on disk.

Preprocessor program processes the code.

Compiler creates object code and stores it on disk.

Linker links the object code with the libraries, creates `a.out` and stores it on disk

Loader puts program in memory.

CPU takes each instruction and executes it, possibly storing new data values as the program executes.

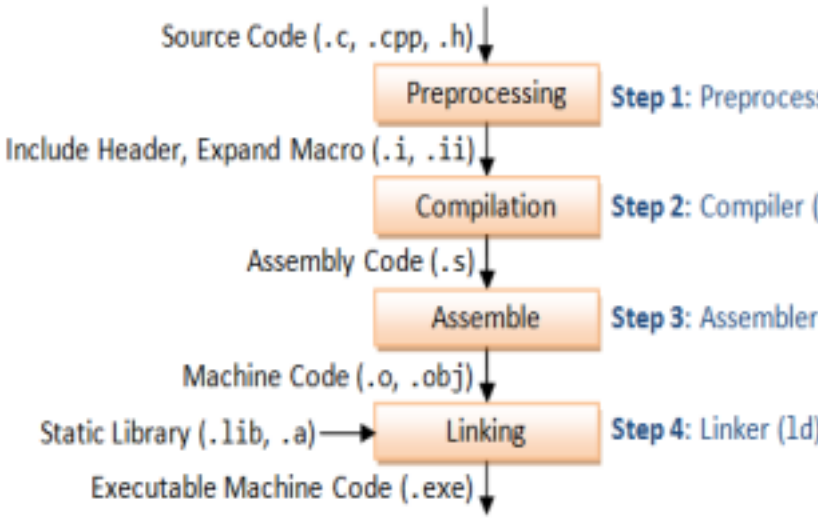
C++ Compilation Process

• Compiler compiles a C/C++ program into executable in 4 steps as shown in the above diagram. For example, a "gcc -o hello.exe hello.c" is carried out as follows:

• **Pre-processing:** via the GNU C Preprocessor (cpp.exe), which includes the headers (#include) and expands the macros (#define).

• The resultant intermediate file "hello.i" contains the expanded source code.

• **Compilation:** The compiler compiles the pre processed source code into assembly code for



a specific processor.

- The resultant assembly file is "hello.s".
- **Assembly:** The assembler (as.exe) converts the assembly code into machine code in the object file "hello.o".
- **Linker:** Finally, the linker (ld.exe) links the object code with the library code to produce an executable file "hello.exe".

C++ Programming Examples

- Following are several examples
  - The examples illustrate many important features of C++
  - Each example is analyzed one statement at a time.

```
1 // Fig. 1.2: fig01_02.cpp
2 // A first program in C++
3 #include <iostream>
4
5 int main()
```

```
6 {

7 std::cout << "Welcome to C++!\n"; 8

9 return 0; // indicate that program ended successfully

10 }
```

```
Welcome to C++!
```

include the contents of the file **<iostream>**, which

C++ programs contain one or more functions, one of

includes input/output operations (such as printing to which must be **main**

the screen).

Parenthesis are used to indicate a function

the computer to perform any action.

preprocessor directive

Message to the C++ preprocessor.

Lines beginning with **#** are preprocessor directives.

Comments

Written between **/*** and ***/** or following a **//**.

Improve program readability and do not cause

#include <iostream> tells the preprocessor to

Prints the *string* of characters contained between the

int means that **main** "returns" an integer value.

quotation marks.

return is a way to exit a function from a function.

More in Chapter 3.

A left brace `{` begins the body of every function

The entire line, including **`std::cout`**, the **`<<`**

and a right brace `}` ends it.

operator, the *string* **`"Welcome to C++!\n"`** and

`return 0`, in this case, means that

the *semicolon* (`;`), is called a *statement*.

the program terminated normally.

All statements must end with a semicolon.

A Simple Program: Printing a Line of Text

- **std::cout**
  - Standard output stream object
  - "Connected" to the screen
  - **std::** specifies the "namespace" which **cout** belongs to
    - **std::** can be removed through the use of **using** statements
- **<<**
  - Stream insertion operator

- Value to the right of the operator (right operand) inserted into output stream (which is connected to the screen)
- **std::cout << “Welcome to C++!\n”;**

- ****
  - Escape character
  - Indicates that a “special” character is to be output

A Simple Program: Printing a Line of Text

Escape Sequence	Description
\n	Newline. Position the screen cursor to the beginning of the next line.
\t	Horizontal tab. Move the screen cursor to the next tab stop.
\r	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
\a	Alert. Sound the system bell.
****	Backslash. Used to print a backslash character.
\"	Double quote. Used to print a double quote character.

- There are multiple ways to print text • Following are more examples

```
1 // Fig. 1.4: fig01_04.cpp
2 // Printing a line with multiple statements
3 #include <iostream>
4
5 int main()
6 {
7 std::cout << "Welcome ";
8 std::cout << "to C++!\n";
9
10 return 0; // indicate that program ended successfully 11 }
```

```
Welcome to C++!
```

Unless new line `'\n'` is specified, the text continues on the same line.

Another Simple Program: Adding Two Integers

- Variables
  - Location in memory where a value can be stored for use by a program
  - Must be declared with a name and a data type before they can be used
  - Some common data types are:
    - **`int`** - integer numbers
    - **`char`** – characters
    - Double – floating point numbers
  - Example: **`int myvariable;`**
    - Declares a variable named **`myvariable`** of type **`int`**

- Example: **`int variable1, variable2;`**
  - Declares two variables, each of type **`int`**

Another Simple Program: Adding Two Integers

- **`>>`** (stream extraction operator)
  - When used with **`std::cin`**, waits for the user to input a value and stores the value in the variable to the right of the operator • The user types a value, then presses the *Enter* (Return) key to send the data to the computer
  - Example:

```
int myVariable;
std::cin >> myVariable;
```

  - Waits for user input, then stores input in **`myVariable`**
- **`=`** (assignment operator)
  - Assigns value to a variable
  - Binary operator (has two operands)
  - Example:

sum = variable1 + variable2;

```
1 // Fig. 1.6: fig01_06.cpp
2 // Addition program
3 #include <iostream>
4
5 int main()
6 {
7 int integer1, integer2, sum; // declaration
8
9 std::cout << "Enter first integer\n"; // prompt
10 std::cin >> integer1; // read an integer
11 std::cout << "Enter second integer\n"; // prompt
12 std::cin >> integer2; // read an integer
13 sum = integer1 + integer2; // assignment of sum
14 std::cout << "Sum is " << sum << std::endl; // print sum
15
16 return 0; // indicate that program ended successfully
17 }
```

Notice how **std::cin** is used to get user input.

std::endl flushes the buffer and prints a newline.

```
Sum is 117
```

```
Enter first integer 45
Enter second integer 72
```

variableName.

Variables can be output using **std::cout <<**

Using declarations

- **using** statements
  - Eliminate the need to use the **std::** prefix
  - Allow us to write cout instead of **std::cout**
  - To use the following functions without the **std::** prefix, write the following at the top of the program

```
using std::cout;
using std::cin;
using std::endl;
```

```
1 // Fig. 1.14: fig01_14.cpp
2 // Using if statements, relational
3 // operators, and equality
operators 4 #include <iostream>
5
6 using std::cout; // program uses
cout 7 using std::cin; // program
uses cin 8 using std::endl; //
program uses endl 9
10 int main()
```

```
11 {
12 int num1, num2;
13
```

Notice the **`using`** statements.

```
14 cout << "Enter two integers, and I will tell you\n"
15 << "the relationships they satisfy: ";
16 cin >> num1 >> num2; // read two integers
17
18 if ( num1 == num2 )
19 cout << num1 << " is equal to " << num2 <<
endl; 20
21 if ( num1 != num2 )
22 cout << num1 << " is not equal to " << num2
<< endl; 23
24 if ( num1 < num2 )
25 cout << num1 << " is less than " << num2 <<
endl; 26
27 if ( num1 > num2 )
28 cout << num1 << " is greater than " << num2
<< endl; 29
30 if ( num1 <= num2 )
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
```

```
31 cout << num1 << " is less than or equal to
" 32 << num2 << endl;
33
```

The **if** statements test the truth of the condition. If it is **true**, body of **if** statement is executed. If not, body is skipped.

```
34 if ( num1 >= num2 )
35 cout << num1 << " is greater than or equal to " 36 <<
num2 << endl;
37
38 return 0; // indicate that program ended successfully 39 }
```

```
3 is less than 7
```

To include multiple statements in a body, delineate them with braces **{ }**.

```
3 is less than or equal to 7
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
```

```
22 is greater than or equal to 12


Enter two integers, and I will tell you
the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

Class Activity#1

- Write a program that generates the following output, using only one *cout statement* and *proper* spacing

1900 2135

1950 2235

2000 2335

2050 2435

Control Structures

- Normally, statements in a program execute one after the other in the order in which they're written.
  - Called sequential execution.
- Various C++ statements enable you to specify that the next statement to execute may be other than the next one in sequence. • Called transfer of control.
- All programs could be written in terms of only three control structures
  - the sequence structure
  - the selection structure and
  - the repetition structure

Control Structures (if else and Switch)

- C++ provides three types of selection statements • The *if* selection statement either performs (selects) an action if a condition (predicate) is true or skips the action if the condition is false.
- The *if*…*else* selection statement performs an action if a condition is true or performs a different action if the condition is false.
- The *switch* selection statement performs one of many different actions, depending on the value of an integer expression.

Control Structures (if else and Switch)

- The *if* selection statement is a single-selection statement because it selects or ignores a *single action* (or, as we'll soon see, a *single group of actions*).
- The *if*…*else* statement is called a double-selection statement because it selects between two different actions (or groups of actions).
- The *switch* selection statement is called a multiple selection statement because it selects among many different actions (or groups of actions).

Control Structures (if selection statement)

- Programs use selection statements to choose among alternative

courses of action.

- For example, whether "student's grade is greater than or equal to 60" is *true* or *false*.

```
if ( grade >= 60 )
cout << "Passed";
```

- If *true*, "Passed" is printed and the next pseudocode statement in order is "performed"
- If *false*, the print statement is ignored and the next statement in order is performed.

Control Structures (if-else double selection statement

- Nested if…else statements test for multiple cases by placing *if…else* selection statements inside other *if…else* selection

statements.

```
if ( studentGrade >= 90 ) // 90 and above gets "A"  cout << "A";
else
 if ( studentGrade >= 80 ) // 80-89 gets "B"
 cout << "B";
 else
 if ( studentGrade >= 70 ) // 70-79 gets "C"  cout << "C";
 else
 if ( studentGrade >= 60 ) // 60-69 gets "D"  cout << "D";
 else // less than 60 gets "F"
 cout << "F";
```

Control Structures (Switch Selection Statement)

- Another alternative to if—else statement is Switch statement.

- The idea behind a switch statement is simple: • an expression (sometimes called the condition) is evaluated to produce a value.
  - If the expression's value is equal to the value after any of the case labels, the statements after the matching case label are executed.
  - If **NO** matching value can be found and a **default label** exists, the statements after the default label are executed instead.

Control Structures (Switch Selection Statement)

```
int main(){
      int num;
      cout<<"Please Enter a number: ";
      cin>>num;
switch (num) {
case 1:
```

```
 cout << 1 << '\n';
 break;
case 2:
 cout << 2 << '\n';
 break;
 case 3:
 cout << 3 << '\n';
 break;
default:
cout<<"Default Case: "<<'\n';
 }
std::cout<<"Switch Terminated: "<<std::endl;
 return 0;}
```

Control Structures (repetition statements)

- C++ provides three types of repetition statements (also

called looping statements or loops) for performing statements repeatedly while a condition (called the loop continuation condition) remains true.

- These are the while, do…while and for statements.
- The *while* and *for* statements perform the action (or group of actions) in their bodies zero or more times.
- The *do…while* statement performs the action (or group of actions) in its body *at least once*.

Control Structures (While repetition Statement)

- A repetition statement (also called a looping statement or a loop) allows you to specify that a program should repeat an

action while some condition remains true.

- The statement contained in the *While* repetition statement constitutes the body of the *While*, which can be a single statement or a block. • Eventually, the condition will become false, the repetition will terminate, and the first pseudocode statement after the repetition statement will execute.

Control Structures (while repetition Statement)

- Consider a program segment designed to find the first power of 3 larger than 100. Suppose the integer variable *product* has been initialized to *3*.
- When the following *while* repetition statement finishes executing, *product* contains the result:
  - *int product = 3;*

```
while ( product <= 100 )
product = 3 * product;
```

Control Structures (Do while Statement)

- A do while statement is a looping construct that works just like a while loop, except the statement always executes at least once.
- After the statement has been executed, the do-while loop checks the condition. If the condition evaluates to true, the path of execution jumps back to the top of the do-while loop and executes it again.

```
do
 statement; // can be a single statement or a compound statement
while (condition);
```

Control Structures (Do while

Statement)

```
#include <iostream>
int main() {
 // selection must be declared outside of the do-while so we can use it later
int selection{};
 do {
 std::cout << "Please make a selection: \n";
 std::cout << "1) Addition\n";
 std::cout << "2) Subtraction\n";
 std::cout << "3) Multiplication\n";
 std::cout << "4) Division\n";
 std::cin >> selection; }
 while (selection != 1 && selection != 2 &&
 selection != 3 && selection != 4);
 // do something with selection here
 // such as a switch statement
 std::cout << "You selected option #" << selection << '\n';
```

```
return 0;
}
```

Control Structures (For Statement)

- The **for statement** (also called a **for loop**) is preferred when we have an obvious loop variable because it lets us easily and concisely define, initialize, test, and change the value of loop variables..
- The for statement looks pretty simple in abstract:

```
for (init-statement; condition; end-expression)
 statement;
```

Control Structures (For Statement)

```
#include <iostream>

int main()
{
 for (int count=0; count <= 10; ++count)
 std::cout << count << ‘ ‘<<i;

 std::cout << '\n';

 return 0;
}
```