Fundamentals of Programming in C++ (CoSc 2031)

BASICS OF PROGRAMMING

Mekonnen K. (MSc)

Email: mekonnen307@gmail.com

1. Process of compiling and running programs

- There are three steps in executing a c++ program: ***Compiling, Linking and Running*** the program.
- The c++ programs have to be typed in a compiler.
- After typing the program, the file is saved with an extension ***.cpp.*** This is known as ***source code.***
- The source code has to be converted to an object code which is understandable by the machine. This process is known as compiling the program.
- You can compile your program by selecting compile from compile menu or press ***Alt+f9***. After compiling a file with the same name as source code file but with extension ***.obj.*** is created.

Cont'd

- ***Second step*** is linking the program which creates an executable file .exe (filename same as source code) after linking the object code and the library files (cs.lib) required for the program.
- In a simple program, linking process may involve one object file and one library file.
- However, in a project, there may be several smaller programs. The object codes of these programs and the library files are linked to create a single executable file.
- ***Third and the last step*** is running the executable file where the statements in the program will be executed one by one.

Programming – Errors

- **Errors** are the mistakes or faults in the program that causes our program to behave unexpectedly and it is no doubt that the well versed and experienced programmers also makes mistakes.
- **Programming error** are generally known as Bugs and the process to remove bugs from program is called as **Debug/Debugging.**
- There are basically three types of error:
  - Compilation error or Syntax error,
  - Runtime error or exception and
  - Logical error.

Cont'd

- **Compilation errors** are the most common error occurred due to typing mistakes or if you don't follow the proper syntax of the specific programming language.
- These error are thrown by the compilers and will prevent your program from running. These errors are most common to beginners. It is also called as *Compile time error or Syntax error*.
- These errors are easy to debug.

Example: Typing int as Int, missing semi colon or paranthesis, type integer for int datatype etc.

Cont'd

- **Run Time errors** (exception) are generated when the program is running and leads to the abnormal behavior or termination of the program. The general cause of Run time errors is because your program is trying to perform an operation that is impossible to carry out.
  - Example: Dividing any number by zero, Accessing any file that doesn't exist etc are common examples of such error.

Cont'd

- **Logical errors** will cause your program to perform undesired operations which you didn't intended your program to perform. These errors occur generally due to improper logic used in program. These types of errors are difficult to debug.
  - Example: Multiplying an uninitialized integer value with some other value will result in undesired output.
- **Linker error**: This error occurs when the files during linking are missing or misspelt.

1.2. Basic syntax and semantics of a high-level languages

The parts of a C++ Program

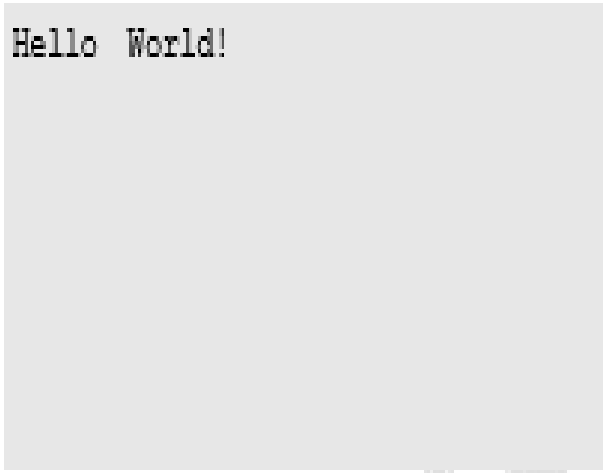
- To understand the basic parts of a simple program in C++, lets have a look at the following code:

```
#include <iostream>

using namespace std;

int main ()
{
        cout << "Hello World!";
        return 0;
}
```

output



Cont'd

- Any C++ program file should be saved with file name extension "**.CPP**".
- Type the program directly into the editor, and save the file as **hello.cpp**, compile it and then run it. It will print the words Hello World! on the computer screen.

#include <iostream>

- Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor.
- In this case the directive #include <iostream> tells the preprocessor to include the iostream standard file.
- This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

using namespace std;

- All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name std. So in order to access its functionality we declare with this expression that we will be using these entities.
- This line is very frequent in C++ programs that use the standard library, and in fact it will be included in most of the source codes included in these tutorials.
- The effects of line 1, i.e. include<iostream.h> is to include the file iostream.h into the program as if the programmer had actually typed it.

int main ()

- int main() indicates the beginning of the program.
- In C++ the parentheses ( ) are used to indicate that the identifier or name on its left is a function. Here main () is also a function.
- Every program in C++ must have only one main() function.
- The word int indicates that main is a value-returning function that should return an integer value. Here int main() means that function main() is a return type function which returns an integer value. In case of a return type function the last statement is return value; .

Cont'd

- Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.
- Every Left French Brace needs to have a corresponding Right French Brace.
- The lines we find between the braces are statements or said to be the body of the function. A statement is a computation step which may produce a value or interact with input and output streams. The end of a single statement ends with semicolon (;).

A brief look at cout and cin

- **cout** and **cin** are declared in the iostream standard file within the **std namespace**, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.
- **Cout** is an object used for printing data to the screen.
- To print a value to the screen, write the word cout, followed by the insertion operator also called *output redirection operator* (<<) and the object to be printed on the screen.
- Syntax: Cout<<Object;
- The object at the right hand side can be:
    - A literal string: "Hello World"
    - A variable: a place holder in memory

Cont'd

- **Cin** is an object used for taking input from the keyboard.
- To take input from the keyboard, write the word cin, followed by the *input redirection operator* **(>>)** and the object name to hold the input value.
- **Syntax**: Cin>>Object;
- Cin will take value from the keyboard and store it in the memory. Thus the cin statement needs a variable which is a reserved memory place holder.
- Both << and >> return their right operand as their result, enabling multiple input or multiple output operations to be combined into one statement. The following example will illustrate how multiple input and output can be performed:

Cont'd

- E.g.: Cin>>var1>>var2>>var3;
- Here three different values will be entered for the three variables. The input should be separated by a space, tan or newline for each variable.
- Cout<<var1<<“,”<<var2<<” and “<<var3;
- Here the values of the three variables will be printed where there is a “,” (comma) between the first and the second variables and the “and” word between the second and the third.

Comments on C++ programs

- **Comments** are parts of the source code disregarded by the compiler. They simply do nothing.
- Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.
- C++ provides two types of comment delimiters:
- **Single Line Comment**: Anything after // {double forward slash} (until the end of the line on which it appears) is considered a comment.
    - Eg: cout<<var1; //this line prints the value of var1
- **Multiple Line Comment**: Anything enclosed by the pair /* and */ is considered a comment.
    - Eg: /*this is a kind of comment where Multiple lines can be enclosed in one C++ program */

Cont'd

- Comments should be used to enhance (not to hinder) the readability of a program. The following two points, in particular, should be noted:
  - A comment should be easier to read and understand
  - Over-use of comments can lead to even less readability.
  - Use of descriptive names for variables and other entities in a program, and proper indentation of the code can reduce the need for using comments.

1.3. Basic Elements of Programming

Identifiers

- The C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).
- C++ does not allow punctuation characters such as @, $, and % within identifiers. C++ is a case-sensitive programming language. Thus, Manpower and manpower are two different identifiers in C++.

Rules for C++ Identifiers

- When naming an identifier, follow these established rules
- **First character:** The first character of the identifier in C++ should positively begin with either an alphabet or an underscore. It means that it strictly cannot begin with a number.
- **No special characters:** C++ does not encourage the use of special characters while naming an identifier. It is evident that we cannot use special characters like the *exclamatory mark* or the "@" symbol.
- **No keywords:** Using keywords as identifiers in C++ is strictly forbidden, as they are reserved words that hold a special meaning to the C++ compiler. If used purposely, you would get a compilation error.

Cont'd

- **No white spaces:** Leaving a gap between identifiers is discouraged. White spaces incorporate blank spaces, newline, carriage return, and horizontal tab.
- **Word limit:** The use of an arbitrarily long sequence of identifier names is restrained. The name of the identifier must not exceed 31 characters, otherwise, it would be insignificant.
- **Case sensitive:** In C++, uppercase and lowercase characters connote different meanings.

Here are some examples of acceptable identifiers

mohd	Piyush	abc	move_name	a_123
myname50	_temp	j	a23b9	retVal

Variables

- **A variable** is a reserved place in memory to store information in.
- A variable will have three components:
- Variables are used for holding data values so that they can be used in various computations in a program.
- All variables have three important properties:
  - **Data Type**: a type which is established when the variable is defined. (e.g. integer, real, character etc.). Data type describes the property of the data and the size of the reserved memory.
  - **Name**: a name which will be used to refer to the value in the variable. A unique identifier for the reserved memory location
  - **Value**: a value which can be changed by assigning a new value to the variable.

Fundamental Variable types.

- Several other variable types are built into C++. They can be conveniently classified as integer, floating-point or character variables.
- Floating-point variable types can be expressed as fraction i.e. they are “real numbers”.
- Character variables hold a single byte. They are used to hold 256 different characters and symbols.
- The type of variables used in C++ program are described in the next table, which lists the variable type.

Fundamental Variable types

Name	Description	Size	Range
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 *or* 4 bytes	1 wide character

- Signed integers are either negative or positive. Unsigned integers are always positive.
- Because both signed and unsigned integers require the same number of bytes, the largest number (the magnitude) that can be stored in an unsigned integer is twice as the largest positive number that can be stored in a signed integer.

Declaring Variables.

- Variables can be created in a process known as declaration.
- **Syntax**: Datatype Variable_Name;
- The declaration will instruct the computer to reserve a memory location with the name and size specified during the declaration.
- Good variable names should be self descriptive .
  - E.g. int myAge; //variable used to store my age
- The name of a variable is called an identifier which should be unique in a program

Initializing Variables

- When a variable is assigned a value at the time of declaration, it is called variable initialization.
- This is identical with declaring a variable and then assigning a value to the variable immediately after declaration.
- The syntax:
  - type identifier = initial_value ;
  - e.g. int a = 0;
  - or: int a; a=0;

Defining a data type (User defined data type)

- The "typedef" Keyword is used to define a data type by the programmer as the user of a programming language is a programmer.
- It is very difficult to write unsigned short int many times in your program if many variables need such kind of declaration.
- C++ enables you to substitute an alias for such phrase by using the keyword typedef, which stands for type definition.
- E.g.:

```
#include<iostream.h>
typedef unsigned short int USHORT;
void main()
{
  USHORT  width = 9;
  ...
}
```

- Immediately after the second line, whenever the compiler encounters the word USHORT it will substitute is with "unsigned short int".

Characters

- ***Characters variables*** (type char) are typically one byte in size, enough to hold 256 different values. A char can be represented as a small number (0 - 255).
- **Char** in C++ are represented as any value inside a single quote. E.g.: ‘x’, ‘A’, ‘5’, ‘a’, etc.
- When the compiler finds such values (characters), it translates back the value to the ASCII values. E.g. ‘a’ has a value 97 in ASCII.

Special Printing characters

- In C++, there are some special characters used for formatting. These are:
  - \n *new line*
  - \t *tab*
  - \b *backspace*
  - \" *double quote*
  - \' *single quote*
  - \? *Question mark*
  - \\ *backslash*
  - \a *alert (beep)*

Constants

- **A constant** is any expression that has a fixed value.
- Like variables, constants are data storage locations in the memory. But, constants, unlike variables their content can not be changed after the declaration.
- Constants must be initialized when they are created by the program, and the programmer can‘t assign a new value to a constant later.
  - E.g.:
  - const int studentPerClass =15;

studentPerClass is a symbolic constant having a value of 15. And 15 is a literal constant directly typed in the program.

defining constants with #define

- Its format is :
  - #define identifier value

    For example:

    #define PI 3.14159

    #define NEWLINE '\
- This defines two new constants: PI and NEWLINE. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:
- In fact the only thing that the compiler preprocessor does when it encounters #define directives is to literally replace any occurrence of their identifier (in the previous example, these were PI and NEWLINE) by the code to which they have been defined (3.14159 and '\n' respectively).

defining constants with the *const* key word

- With the const prefix you can declare constants with a specific type in the same way as you would do with a variable:
  - const int pathwidth = 100;
  - const char tabulator = '\t';
- Here, pathwidth and tabulator are two typed constants. They are treated just like regular variables except that their values cannot be modified after their definition.

Expressions and Statements

- In C++, a statement controls the sequence of execution, evaluates an expression, or does nothing (the null statement).
- All C++ statements end with a semicolon. E.g.: x = a + b; The meaning is: assign the value of the sum of a and b to x.
- **White spaces**: white spaces characters (spaces, tabs, new lines) can't be seen and generally ignored in statements. White spaces should be used to make programs more readable and easier to maintain.
- **Blocks:** a block begins with an opening French brace ({) and ends with a closing French brace (}).
- **Expressions:** an expression is a computation which yields a value. It can also be viewed as any statement that evaluates to a value (returns a value).
- E.g.: the statement 3+2; returns the value 5 and thus is an expression.

Expressions and Statements

- **Expressions: Expression**: An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.
- an expression is a computation which yields a value. It can also be viewed as any statement that evaluates to a value (returns a value).
- E.g. 1: the statement 3+2; returns the value 5 and thus is an expression.

E.g.2: `x = a + b;`

`y = x = a + b;`

Operators

- An operator is a symbol that makes the machine to take an action.
- Different Operators act on one or more operands and can also have different kinds of operators.
- C++ provides several categories of operators, including the following:
  - Assignment operator
  - Arithmetic operator
  - Relational operator
  - Logical operator
  - Increment/decrement operator and
  - Comma operator

Assignment operator (=)

- **The assignment operator** causes the operand on the left side of the assignment statement to have its value changed to the value on the right side of the statement.
- Syntax: Operand1=Operand2;
- Operand1 is always a variable
- Operand2 can be one or combination of:
  - A literal constant: Eg: x=12;
  - A variable: Eg: x=y;
  - An expression: Eg: x=y+2;

Compound assignment operators (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=)

- When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

expression	is equivalent to
`value += increase;`	`value = value + increase;`
`a -= 5;`	`a = a - 5;`
`a /= b;`	`a = a / b;`
`price *= units + 1;`	`price = price * (units + 1);`

Arithmetic operators (+, -, *, /, %).

- Except for remainder or modulo (%), all other arithmetic operators can accept a mix of integers and real operands.
- Generally, if both operands are integers then, the result will be an integer. However, if one or both operands are real then the result will be real.
- When both operands of the division operator (/) are integers, then the division is performed as an integer division and not the normal division we are used to.

Cont'd

- Integer division always results in an integer outcome.
- Division of integer by integer will not round off to the next integer

  E.g.: 9/2 gives 4 not 4.5

  -9/2 gives -4 not -4.5

- To obtain a real division when both operands are integers, you should cast one of the operands to be real.

  E.g.:
  ```
  int cost = 100;
  Int volume = 80;
  Double unitPrice = cost/(double)volume;
  ```

- The module(%) is an operator that gives the remainder of a division of two integer values.

  E.g.: a = 11 % 3 a is 2

Relational operator (==, !=, > , <, >=, <=)

- In order to evaluate a comparison between two expressions, we can use the relational operator.
- The result of a relational operator is a bool value that can only be true or false according to the result of the comparison.
- E.g.: (7 = = 5) would return false or returns 0
- (5 > 4) would return true or returns 1
- The operands of a relational operator must evaluate to a number.
- Characters are valid operands since they are represented by numeric values.
- For E.g.: 'A' < 'F' would return true or 1. it is like (65 < 70

Logical Operators (!, &&, | |)

- **Logical negation (!)** is a unary operator, which negates the logical value of its operand. If its operand is non zero, it produce 0, and if it is 0 it produce 1.
- **Logical AND (&&)** produces 0 if one or both of its operands evaluate to 0 otherwise it produces 1.
- **Logical OR (||)** produces 0 if both of its operands evaluate to 0 otherwise, it produces 1.
- E.g.: !20 //gives 0 which mean false

  10 && 5 //gives 1 means true

  10 || 5.5 //gives 1 which mean true

  10 && 0 // gives 0 which mean false

N.B. In general, any non-zero value can be used to represent the logical true, whereas only zero represents the logical false.

Increment/Decrement Operators: (++) and (--)

The auto increment (++) and auto decrement (--) operators used to add and subtract 1 from numeric variable

E.g.

```
int a=9;

a++; //a =10 we can also write as ++a;

int b=12;

b--; //b=11 we can also write as ++b;
```

Cont'd

Prefix and Postfix

- The prefix type is written before the variable.

  Eg (++ myAge), whereas the postfix type appears after the variable name (myAge ++).

- Prefix and postfix operators can not be used at once on a single variable: Eg: ++age-- or --age++ or ++age++ or - - age - - is invalid
- In a simple statement, either type may be used. But in complex statements, there will be a difference.
- The prefix operator is evaluated before the assignment, and the postfix operator is evaluated after the assignment.

Comma Operator (,)

- Multiple expressions can be combined into one expression using the comma operator.
- The comma operator takes two operands. Operand1,Operand2
- The comma operator can be used during multiple declaration, for the condition operator and for function declaration, etc
- It the first evaluates the left operand and then the right operand, and returns the value of the latter as the final outcome.

E.g.

```
int m,n,min;
int mCount = 0, nCount = 0;
min = (m < n ? (mCount++ , m) : (nCount++ , n));
```

- Here, when m is less than n, mCount++ is evaluated and the value of m is stored in min. otherwise, nCount++ is evaluated and the value of n is stored in min.

Operator Precedence

- The order in which operators are evaluated in an expression is significant and is determined by precedence rules.
- Operators in higher levels take precedence over operators in lower levels.
- Precedence Table:

Level	Operator	Order
Highest	++ -- () (post fix)	Right to left
	sizeof() ++ -- (prefix)	Right to left
	* / %	Left to right
	+ -	Left to right
	< <= > >=	Left to right
	== !=	Left to right
	&&	Left to right
	||	Left to right
	? :	Left to right
	= ,+=, -=, *=, /=,^= ,%=, &= ,|= ,<<= ,>>=	Right to left
	,	Left to right

Cont'd

- E.g.
  - ✓ a = = b + c * d is evaluated first because * has a higher precedence than + and = =.
  - ✓ The result is then added to b because + has a higher precedence than = = And then == is evaluated.
- Precedence rules can be overridden by using brackets. E.g. rewriting the above expression as: a = = (b + c) * d causes + to be evaluated before *.
- Operators with the same precedence level are evaluated in the order specified by the column on the table of precedence rule. E.g. a = b += c the evaluation order is right to left, so the first b += c is evaluated followed by a = b.

The End!!!