vii. System maintenance is performed in response to _____.

a) Business changes
b) Hardware and software changes
c) User's requests for additional features
d) All of the above

Q.3 Write TRUE/FALSE against the following statements.

i. A stakeholder is any person or organization who has a direct or indirect interest in the project.
ii. System maintenance is performed to gather the requirements of the proposed system.
iii. The how, when, what type of questions are answered in the planning phase of SDLC.
iv. The three architectures i.e. algorithms, pseudo code and flowcharts are used in the implementation phase of SDLC.
v. Each phase of SDLC is accomplished as a discrete, separate step.

Q.4 Define software development life cycle (SDLC). What are its objectives?

Q.5 What is a system? Where exactly the testing activities begin in SDLC?

Q.6 Why software development life cycle is important for the development of software?

Q.7 Who are stakeholders of SDLC? Describe their responsibilities.

Q.8 What is meant by the term software requirement? Differentiate between functional and non-functional requirements.

Q.9 Design a flowchart for the following algorithm.

```
sum = 0, N=5
x = 1
while x ≤ N do
sum = sum + x
x = x + 1
end while
print sum
```


UNIT 3

PROGRAMMING USING C++

After the completion of Unit-3, Students will be able to

- Define Program.
- Define header file and reserved words.
- Describe the structure of a C++ program.
- Know the use of statement terminator (;).
- Explain the purpose of comments and their syntax.
- Differentiate between Constant and Variable.
- Explain the rules for specifying variable names.
- Know different Data types in C++ and their size in bytes.
- Define constant qualifier- const.
- Explain the process of declaring and initializing variables
- Use type casting.
- Explain the use of **cout** and **cin** statements.
- Define **getch**(), **gets**() and **puts**() functions.
- Define and use escape sequences (\a, \b, \r, \t, \\, \', \").
- Make use of commonly used I/O handling functions.
- Use manipulators **endl** and **setw**.
- Define and use different types of operators in C++.
- Identify Unary, binary and ternary operators.
- Define and explain expression and compound expression.
- Explain the order of precedence of operators.

3.1 INTRODUCTION

This Unit is about the introduction to Object Oriented Programming Language, C++, and covers its basics in detail. The Unit is focused on basic C++ program structure, variables used in C++, input/output statements and the use of operators in the language.

Bjarne Stroustrup at Bell Labs initially developed C++ during the early 1980's. It was designed to support the features of C such as efficiency and low-level support for system level coding. Added to this were features such as classes with inheritance and virtual functions, derived from the Simula language, and operator overloading, derived from Algol. C++ is best described as a superset of C, with full support for object-oriented programming. This language is in wide spread use. The first commercial release of the C++ language was in October of 1985.

3.1.1 Computer program

A computer program is a set of instructions written in computer languages to perform a specified task for a computer. A computer program tells the computer that what to do and in which order to do. Different types of programming languages are used to develop programs. Some commonly used programming languages are C++, JAVA, SQL, HTML, etc.

3.1.2 Header files and Reserved words

Header files and reserved words are the two important components of almost every C++programs.

a. Header files

Header files also known as **include files,** are standard library files that have an extension of (.h) and which are used to hold declarations for other files.

Consider the following program:

```
// header file example
#include <iostream.h>
int main()
{
cout << "Hello, world!" << endl;
return 0;
}
```

Output of the program

Hello, world!

This program prints the string "Hello, world!" to the screen using *cout*. However, this program never defines *cout*, so how does the compiler know about the object *cout*? The answer for this is that *cout* has been declared in a header file called "*iostream*". When the line *#include <iostream.h>* is used in the program, the compiler locates and read all the declarations from a header file named "*iostream*".

b. Reserved words

Reserved words or keywords are those words which have their special meaning within the C++ language and are reserved for some specific purpose. C++ reserved words cannot be used for any other purpose in a C++ program and even cannot be used as variables. Here is a list of C++ keywords shown in Table 3.1.

asm	auto	bool	break	case	catch	char	class
const	const_cast	continue	default	delete	do	double	dynamic_cast
else	enum	explicit	export	extern	false	float	for
friend	goto	if	inline	int	long	mutable	namespace
new	operator	private	protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast	struct	switch	template
this	throw	true	try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile	wchar_t	while	

Table 3.1: List of C++ Keywords

3.1.3 Structure of a C++ program

General syntax of a C++ program is given below.

```
Preprocessor directives <header file>
int main()
{
Body of the program
}
```

Each C++ program has three main components. These are: **Preprocessor directives**, **main** function and body of the **main** function. Now, consider the following "Hello World" example to explain these components.

```
#include <iostream.h>
#include <conio.h>
int main()
{
cout << "Hello World!";
getch();
return 0;
}
```

Output of the Program

```
Hello World!
```

The result of the above program is that it prints "Hello World!" on the screen. It is one of the simplest programs that can be written in C++, but it contains the fundamental components of almost every C++ program.

a. Preprocessor directives (#include, #define)

A preprocessor is a collection of special statements which are executed before the compilation process.

Almost every C++ program contains a preprocessor directive. The *#include* preprocessor directives is commonly used to insert *header files* with extension '.h'. These header files are already stored in computer in *include* directory. In the above program, two *#include* directives have been used, *#include<iostream.h>* and *#include<conio.h>*. *#include<iostream.h>* is used for the C++ object *cout* and *#include<conio.h>* for the *built-in* function *getch()*.

In C++, the other commonly used preprocessor directive is *#define* which is used to define symbolic constants. Its general format is:

***#define* identifier value**

For example:

```
#define PI 3.14159
#define NEWLINE '\n'
```

This defines two new constants: PI and NEWLINE. Once they are defined, they can be used in the rest of the program as if they were any other regular constant, for example:

```
#include <iostream.h>
#include <conio.h>
#define PI 3.14159
#define NEWLINE '\n'
int main()
{
```

```
double r=5.0; // radius
double circle;
circle = 2 * PI * r;
cout << "Area of the Circle: "<<circle;
cout << NEWLINE;
getch();
return 0;
}
```

Output of the Program

Area of the Circle: 31.4159

The *#define* directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (;) at its end.

b. main( ) Function

The *main* function is the point by where all C++ programs start their execution. It does not matter whether there are other functions with other names defined before or after it, the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a *main* function.

The word *main* is followed in the code by a pair of parentheses ( ).

c. Body of the main( ) function

After the parentheses of the *main* function, the body of *main* function starts which is enclosed in braces { }. When this function is executed, the instructions written within it perform their task. Inside the body of *main* function, C++ statements are written.

C++ Statements

A C++ *statement* is a simple or compound expression that can actually produce some effect. It is actually an instruction to the computer for taking an action.

For example *cout << "Hello World!";* is a C++ statement. The following are some examples of statements.

cout << "Hello World!";

In fact, this statement performs the only action of displaying the result "Hello World!" on the screen.

cout is the name of the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream. *cout* is declared in the *<iostream.h>* standard header file.

getch ();

This is a standard library function defined in the header file *<conio.h>* and is used to wait for the user to input a character from the keyboard.

return 0;

The return statement causes the *main* function to return control to the operating system. Return may be followed by a return code. The code 0 returned by the *return* statement tells the operating system that the program terminates normally.

The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines. For example, the following program can also be written in a single line.

```
int main()
{
```

```
cout << " Hello World!";
return 0;
}
```

Its single line version will be:

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

3.1.4 Statement terminator (;)

Each C++ statement is terminated by a symbol named semicolon (;) which is called *statement terminator*. In C++, the separation between statements is specified with this ending semicolon (;) at the end of each statement. It does not matter to write more than one statement on a single line but matter if you do not separate them with semicolons. The use of each statement on a separate line is only to add clarity in the program.

3.1.5 Comments and their syntax in C++

Comments are parts of the source code ignored by the compiler. They do nothing but simply increase the readability and understandability of a program. Their purpose is only to allow the programmers to insert notes or descriptions within the source code.
C++ supports two types of comments:

a. Single line comment

It is represented by the double slash symbol //. It discards everything from where the pair of slashes signs // is found up to the end of the same line.

b. Multiline comment

These types of comments are represented by the symbols /* ..... */. It ignores everything between the /* characters and the first appearance of the */ characters, with the possibility of including more than one line.

Consider the following program to demonstrate comments.

```
/* program to demonstrate comments in C++*/
#include <iostream.h>
#include<conio.h>
int main()
{
cout << "Welcome! "; // prints Welcome!
cout << "I'm a C++ program demonstrating comments";
getch();
return 0;
}
```

Output of the Program

```
Welcome!
I'm a C++ program
```

In the above example, the first two lines are multi-line comments while the comment at line 7 is a single line comment.
Comments are always ignored by the compiler when it compiles the program. If comments are included within the source code of your programs without using // or /*, the compiler will take them as if they were C++ expressions.

3.2. C++ Constants and Variables

Variables and constants are the important components of every programming language.

3.2.1 Constants and Variables

a. Constants

Constants are the expressions which have fixed value. Constants are generally categorized into: Literal, Boolean and Symbolic constants. They are used to express particular values within the source code of a program.

Consider the following line of code:

VarA = 555;

In this line of code, 555 is a **literal constant.** The literal constants are further categorized into the following types.

- String constants
- Numeric constants
- Character constants

i. String constants

A string constant is a sequence of characters enclosed in double quotation marks is called string constant. Its maximum length is 256 characters. Following are some examples of valid string constants.

"Welcome to the first C++ Program"

"The result="

ii. Numeric constants

Numeric constants consist of positive or negative signed numerals. There are four types of numeric constants. These are:

- Integer constant
- Floating point constant
- Hexadecimal constant
- Octal constant

- ***Integer constants***: Integer constants are those positives or negative signed numbers that do not contain a decimal point e.g. 2010, -321 etc.

- ***Floating point constants:*** The numeric constants having a decimal point are called floating point constants e.g. 33.55 and -0.22 etc. These constants can also be either positive signed or negative signed. These constants can also be represented in its exponential form by the use of alphabet 'e' or 'E' to denote the exponents of the numbers. For example.

  9010E10

  7810.11E-11

  -10.990e8

  -1.001e-1

- ***Hexadecimal constants:*** The constants represented in hexadecimal number system are called hexadecimal constants. To represent a hexadecimal constant, begin the constant with 0x or 0X, followed by a sequence of digits in the range 0 through 9 and a through f (or A through F). The digits a to f (or A to F) represent values in the range 10 through 15. For example.

  int i = 0x3fff; // Hexadecimal constant

  int j = 0X3FFF; // Hexadecimal constant

- ***Octal constants:*** The representation of constants in octal number system is called octal constants. To specify an octal constant, begin the specification with 0, followed by a sequence of digits in the range 0 through 7. Consider the following example:

  int i = 0377; // Octal constant

  int j = 0397; // Error: 9 is not an octal digit

Octal numbers are integer numbers of base 8 and their digits are 0 to 7.

iii. Character constants

A character enclosed within single quotes is called character constant.
Consider the following examples:

'A', 'a', '.', '?'

b. Variables

A variable in C++ is a name for a piece of memory that can be used to store information. A variable can be thought as a mailbox where information can be put and retrieved from. All computers have memory, called Random Access Memory (RAM) that is available for programs to use. When a variable is declared, a piece of that memory is set aside for that variable.

Consider the following segment of code.

```
int a = 5;
int b = 2;
int result = a - b;
```

In this segment, a, b and result are the integer variables that reside in RAM and store integer values.

3.2.2 Rules for naming variables

The symbol used for a variable is called an identifier or variable name. A set of rules are used for naming variables. These rules are:

- Valid identifier is a sequence of one or more letters, digits or underscores characters (_).

- Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid.
- Variable identifiers always have to begin with a letter. They can also begin with an underscore character(_).
- In no case, a variable can begin with a digit.
- A Reserved Word of the C++ language cannot be used as an identifier.
- Names should be meaningful.
- Names should not be too long.

```
#include <iostream>
#include<conio.h>
using namespace std;
using namespace std
```

C++ language is a *"case sensitive"* language which means that an identifier written in capital letters is not equivalent to the one written in small letters.

3.2. 3 Declaration and Initialization of variables

a. Declaration of variables

In order to use a variable in C++, we must first declare it. The declaration of a variable specifies that which type of data this variable will store/use. The syntax to declare a new variable is to write the specifier of the desired data type such as *int, bool, float* etc. followed by a valid identifier. For example:

Data type variable_name;

int a;

When this statement is executed by the CPU, a piece of memory from RAM will be set aside and it will be named as a. More than one variables of the same type can be declared in a single statement by separating them with commas. For example:
int a, b, c;

This line of code declares three variables *a*, *b* and *c*, all of type *int*. This declaration is exactly the same as shown in the following segment of code:

```
int a;
int b;
int c;
```

Consider the following example to see the use of variable declaration.

```
#include <iostream.h>
#include <conio.h>
int main()
{
// declaring variables:
int a, b;
int result;
// process:
a = 55;
b = 22;
result = a - b;
// print out the result:
cout << "The Result is= "<<result;
// terminate the program:
getch();
return 0;
}
```

Output of the Program

The Result is= 33

The declaration of the variables is not necessary at the start of the program. A variable can be declared in the program where it is needed. Consider the following example.

```
// variables declaration at the point of application
#include<conio.h>
#include<iostream.h>
int main()
{
cout << "Enter a number: ";
int x; // we need x starting here.
cin >> x;
cout << "Enter another number: ";
int y; // we don't need y until now
cin >> y;
cout << "The sum is: " << x + y << endl;
getch();
return 0;
}
```

Output of the Program

Enter a number: 4
Enter another number: 10
The sum is: 14

b. Initialization of variables

Assigning values to a variable during declaration time is called *initialization*. When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable.

Consider the following program to demonstrate the initialization of variables.

```
#include<conio.h>
#include <iostream.h>
int main()
{
int Vara=55; // initial value = 55
int Varb(22); // initial value = 22
int result; // initial value undetermined
result = Vara - Varb;
cout << "The Result is= "<<result;
getch();
return 0;
}
```

Output of the Program

The Result is= 33

3.2. 4 Fundamental data types in C++

While programming, we store the variables in computer's memory, but the computers have to know that what kind of data we want to store in them.

Data type tells the compiler that what types of data, means integer, character or floating point etc. the variable will store and what will be the range of the values.

The most commonly used data types in C++ are:

- Character
- Integer
- Floating point

- Boolean
- Unsigned

a. Character data type

The *char* keyword represents the character data type in C++. Any character belonging to the ASCII character set is called a character data type. The maximum size of *char* data type is 8 bits (1 byte). *Signed char* and *unsigned char* are its two distinct types. They occupy the same amount of memory space. Consider the following line of code:

char x;

Here, x is a character type variable that occupies 1 byte of memory. Variables declared as **char** data type are used to store character constants. Character constants are always represented with single quotes. For example:

```
char gender;
gender= 'M';
```

Here, gender is a character type variable that occupies 1 byte of memory and store character constant, M, which stands for Male.

b. Integer data types

The data types which store only integer numbers such as 100, -200 etc. are called integer data types. Its sub types are:

- int
- short int
- long

i. int data type

The keyword int stands for the integer data type in C++ and normally has two bytes of memory (16 bits). A 16 bit integer may fall in the range of -2^{15} to $2^{15}-1$, which means that it can store values in the range of -32768 to 32767.

ii. short int data type

short int is used to declare the short integer data type in C++. The maximum size of the *short int* data type is 2 bytes (16 bits). It may fall in the range -2^{15} to $2^{15}-1$, which means that it can store the values from –32768 to 32767.

iii. long int data type

long int stands for long integer data type and is used in C++ to store larger integer values. Its size is 4 bytes (32 bits) which means that it can stores values in the range of –2147483648 to 2147483647.

c. Floating point data type

Integers are only used for storing whole numbers, but sometimes we need to store very large numbers, or numbers with a decimal point. The data types which are used to store such type of data are called **floating point** data types. Variables of floating point types hold real numbers, such as 4.0, 2.5, 3.33, or 0.1226. There are three different types of floating point data types:

- float
- double
- long double

i. float data type

In C++, the *float* data type is used to declare floating point type of variables to store numbers with decimal point. It needs 4 bytes (32 bits) of memory to store values.

ii. double data type

double is a keyword to represent double precision floating point numbers in C++. *double* data type occupies 8 bytes (64 bits) of memory. The size of *double* data type is a rational number in the same range as long float and is stored in the form of floating point representation with binary mantissa and exponent.

iii. long double data type

In C++, the *long double* data type is used to declare *long double* type of variables to represent long double precision floating point numbers in C++. It needs 10 bytes (80 bits) of memory space in the RAM.

d. Boolean data type

bool is the keyword used to represent Boolean data type. It is capable of holding one of the two values: true (1) or false (0). It occupies 1 byte of memory.

bool flag;

e. Unsigned data type

The data types discussed so far are signed data types which means that one bit is reserved for the sign of the value (i.e. +/-). The unsigned numbers are whole numbers and always hold positive values starting from 0 till its maximum size. Here, the sign bit also used for the value which means no bit is reserved for holding a sign (+/-). The unsigned numbers may be classified into four types:

- unsigned char
- unsigned integer
- unsigned short integer
- unsigned long integer

A summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one is given in Table 3.2.

Data Type	Name	Description	Size	Range
character	char	Character or small integer	1byte	signed: -128 to 127
				unsigned: 0 to 255
Integer	short int (short)	Short Integer	2bytes	signed: -32768 to 32767
				unsigned: 0 to 65535
	int	Integer (16 bit system)	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
		Integer (32 bit system)	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
	long int (long)	Long integer	4bytes	signed: -2147483648 to 2147483647
				unsigned: 0 to 4294967295
Floating Point	float	Floating point number	4bytes	3.4×10^{-38} to 3.4×10^{38}
	double	Double precision floating point number	8bytes	1.7×10^{-308} to 1.7×10^{308}
	long double	Long double precision floating point number	10bytes	1.7×10^{-4932} to 1.7×10^{4932}
Boolean	bool	Boolean value. It can take one of two values: true or false.	1byte	true (1) or false (0)

Table 3.2: C++ Data Types

3.2.5 Constant qualifier

const keyword is used to define constant identifier whose values remain constant during the whole life of the program. *Const* identifier must be assigned with a value when declared, and then that value cannot be changed. Here is the segment of code used to define constant using *const* identifier keyword.

```
const int DollarRate = 86;
```


Declaring a variable as **const** prevents the programmers from unintentionally changing its value. Consider the following lines of code to change the value of variable DollarRate.

```
const int DollarRate = 86;
DollarRate = 123; // compiler error!
```

A compiler error message will be displayed by the compiler because the code at line 2 tries to change the value of the variable DollarRate from 86 to 123, which is declared as constant. It is an alternative to **#define** preprocessor directive which is used for defining constants in a program.

3.2.6 Type casting operator

The conversion of data from one type to another is called **type casting**. It has two types.

- Implicit type casting
- Explicit type casting

In **Implicit type casting**, the compiler automatically converts data from one type to another when they are used in expressions. When a value of one type is assigned to another type, the compiler implicitly converts that value into the value of the new type. For example:

```
double a= 3; // implicit casting to double value 3.0
int b= 3.14156; // implicit casting to integer value 3
```

In C++, there are several ways to convert one type to another but the simplest one is to precede the expression to be converted by the new type enclosed in parentheses ( ). Such type of conversion is called **explicit type casting**.

Consider the following segment of code:

```
int x;
float pi = 3.14;
x = (int) pi;
```

In the above code, the float number 3.14 is converted to an integer value 3 and the remainder is discarded. Here, the type casting operator is (**int**). Another way for explicit type casting is to precede the expression to be converted by the type and enclose the expression in parentheses. Consider the following line of code:

```
x = int ( pi );
```

3.3 Input/Output Handling

For input/output operations, C++ uses *streams* in sequential media such as the screen or the keyboard. A stream is an object where a C++ program inserts characters to or extracts the characters from stream.

All the standard input and output stream objects are declared within the header file **iostream** which is a part of the standard C++ library.

3.3.1 Standard output (cout)

The default standard output of a C++ program is the screen, and **cout** is the C++ stream object which defines it. **cout** is used in conjunction with the *insertion* operator, which is written as <<. Consider the following segment of code:

```
cout << "Output"; // prints Output on the screen
cout << 121; // prints number 121 on the screen
cout << var; // prints the content of 'var' on the screen
```


The insertion operator << inserts the data that follows it into the stream. In the above examples, it inserts the string "Output", the numerical constant 121 and the value of the variable var into the standard output stream **cout**.

The insertion operator << may be used more than once in a single statement. Consider the following line of code.

```
cout << "Welcome " << "to " << "the first C++ statement";
```

This statement prints the message "Welcome to the first C++ statement" on the screen as an output. The repetition of the insertion operator << is needed when more than one variables or combination of variables and constants are needed to print. Consider the following line of code.

```
cout << "Assalamoalaikum, I am a student of " << class << " Class";
```

If the value of the variable class1 is 12 then the above statement would be:
Assalamoalaikum, I am a student of 12 Class

3.3.2 Standard input (cin)

The standard input device used for entering input to the computer is keyboard. In C++, the stream *extraction* operator >> and object **cin** are used to handle the standard input. The operator >> must be followed by the variable that will store the data that is going to be extracted from the stream. Consider the following lines of code:

```
int marks;
cin >> marks;
```

Here, the first statement, *int* marks; declares a variable of type **int** called 'marks', and the second statement waits for an input from **cin** in order to store it in this integer variable.

The object **cin** can only process the input from the keyboard once the enter key is pressed. Therefore, even if you request a single character, the extraction from **cin** will not process the input until the user presses enter after the character has been introduced.

Consider the following example to demonstrate **cin** and **cout** objects.

```
// program to demonstrate cin and cout objects
#include <iostream.h>
#include<conio.h>
int main()
{
int i;
float age;
cout << "Please enter your age: ";
cin >> age;
cout << "The age you entered is " << age <<" years";
getch();
return 0;
}
```

Output of the Program

```
Please enter your age:
16
The age you entered is 16 years
```

cin can also be used to request more than one input from the user.

Consider the following line of code:

```
cin >> x >> y;
```

is equivalent to:

```
cin >> x;
cin >> y;
```


Here, in both the cases, the user need to enter two values, one for variable x and another one for variable y.

- cin and strings

cin can also be used to get strings with the extraction operator >> in the same way as it is used to get values for variables. Consider the following line of code:

```
cin >> TestStr;
```

cin object stops reading characters when it gets any blank space character, so in this case of entering strings, the object will get only one word. For example, if one wants to input a sentence using **cin** object, it will only accept the first word of the sentence and will discard the rest of the words.

In order to get entire line with spaces between the words, the function **gets** is used which is recommended by the programmers as compared to **cin**. Consider the following program demonstrating **cin** object for string.

```
// cin with strings
#include <iostream.h>
#include<conio.h>
int main()
{
char TestStr[60];
cout << " Enter your name please \n";
cin >>Test Str;
cout << "\nYour name is" << TestStr;
getch();
return 0;
}
```

Output of the Program

Enter your name please
Mohammad Khalid
Your name is Mohammad Khalid

3.3.3 gets(), puts() and getch() functions

C++ uses a wide range of standard Input/output functions. These functions are defined in standard library header files. The following sub-sections discuss some of the most commonly used I/O built-in functions.

a. gets() function

gets() is a standard library function defined in **stdio.h** header file which is used to get a string from the keyboard. Its general syntax is:

gets(string variable name);

It reads characters from stdin and stores them as a string into 'string variable' until a newline character or the EOF is reached. Consider the following example:

```
/* gets example */
#include <stdio.h>
#include <conio.h>
#include<iostream.h>
int main()
{
char Test Str [150];
cout<<"Insert your full address: ";
gets (TestStr);
cout<<"\nYour address is:"<<TestStr;
```



```
getch();
return 0;
}
```

Output of the Program

Insert your full address: *H.No. 10 Lalazar Colony*
Peshawar
Your address is: *H.No. 10 Lalazar Colony*
Peshawar

b. puts() function

puts() function writes a string to the screen and appends a newline character automatically at the string. The general syntax of puts () function is:

puts (string variable);

Consider the following example to demonstrate **gets()** and **puts()** functions:

```
#include <stdio.h>
#include<conio.h>
#include<iostream.h>
int main()
{
char TestStr [100];
cout<< " Enter a string\n";
gets(TestStr);
cout<< "You entered: ";
puts (TestStr);
getch();
return 0;
}
```

Output of the Program

Enter a string

Hello World

You entered: *Hello World*

c. getch() function

getch() function is a function that obtains the next available keystroke or character from the console. As a result of this function, nothing is echoed on the screen. When no keystroke is available, the function waits until a key is pressed. The value of the keystroke is returned by the function when a key is pressed. Its general syntax is:

getch( );

Consider the follow ing C++ program that demonstrates the use of **getch**() function.

```
/* getch() example */
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
int main()
{
char ch;
cout<< "Press any key\n";
ch = getch();
cout<<"You pressed: " << ch <<endl;
getch();
return 0;
}
```

Output of the Program

Press any key

S

You pressed: S

3.3.4. Escape sequences

C++ has some characters that have special meaning. These characters are called **escape sequences**. An escape sequence starts with a \ followed by a letter or number.

Table 3.3 shows the list of all escape sequences that are used in C++ along with their purpose.

Name	Symbol	Meaning
Alert	\a	Makes an alert, such as a beep
Backspace	\b	Moves the cursor back one space
Formfeed	\f	Moves the cursor to next logical page
Newline	\n	Moves cursor to next line
Carriage return	\r	Moves cursor to beginning of line
Horizontal tab	\t	Prints a horizontal tab
Vertical tab	\v	Prints a vertical tab
Single quote	\'	Prints a single quote
Double quote	\"	Prints a double quote
Backslash	\\	Prints a backslash
Question mark	\?	Prints a question mark
Octal/hex number	\(number)	Translates into char represented by octal/hex number

Table 3.3: Escape Sequences

The most common escape sequence is \n, which can be used to embed a newline in a string of text:

```
// \n escape sequence program
#include <iostream.h>
#include<conio.h>
int main()
{
cout<< "Welcome to \nthe College " <<endl;
getch();
return 0;
}
```

Output of the Program

Welcome to
the College

Consider another program for escape sequence character "\t".

```
// escape sequence \t program
#include <iostream.h>
#include<conio.h>
int main()
{
cout<< "First part\tSecond part";
getch();
return 0;
}
```

Output of the program

First part Second part

3.3.5 I/O handling functions

Most commonly used

- **getchar()** stands for get character and is used to get a single character from standard input device (keyboard). **getchar**() is defined in **stdio.h** header file. The general syntax for the use of getchar () function is:

```
int getchar ();
```

This function gets no parameters.

- **putchar()** is a standard output function defined in **stdio.h** header file and is used to display a single character, received as an argument, on the screen. Its general syntax is:

```
putchar (ch); // where ch is a char type variable holding a character constant
```

This putchar() display the character constant stored in 'ch' on the screen. Consider the following example:

```
// getchar() and putchar() example
#include <conio.h>
#include <stdio.h>
#include<iostream.h>
int main()
{
char ch;
cout<<"Enter character for gender\n";
ch=getchar();
cout<< "You have entered: ";
```

```
putchar (ch);
getch();
return 0;
}
```

Output of the Program

Enter character for gender
M
You have entered: M

This program reads a character from the keyboard using **getchar**() function and assign it to a character type variable 'ch' which is displayed on the screen by another standard library function **putchar**().

3.3.6 endl and setw Manipulator

Manipulators are used for formatting output. The data is manipulated by the programmer's choice of display. There are several manipulators available in C++. The most commonly used manipulators are discussed below:

a. endl manipulator

This manipulator has the same functionality as the '\n' newline character but with the difference of flushing the stream. Consider the following example:

```
// endl manipulator program
#include <iostream.h>
#include <conio.h>
int main()
{
cout << "Welcome" << endl;
cout << "to" << endl;
```



```
cout << "the College";
getch();
return 0;
}
```

Output of the program

Welcome
to
the College

b. setw manipulator

This manipulator sets the minimum field width on output. The syntax is:

setw(x)

Here, **setw** causes the number or string that follows it to be printed within a field of x characters wide and x is the argument set in **setw** manipulator. The header file that must be included while using **setw** manipulator is **<iomanip.h>**. Consider the following example:

```
//setw manipulator program
#include <iostream.h>
#include <iomanip.h>
#include<conio.h>
int main( )
{
int x1=2,x2= 4, x3=6;
cout<<setw(8)<<"Number"<<setw(20)<<"Square"<<endl
<< setw(8) << "2" << setw(20)<< x1*x1 << endl
```

```
<< setw(8) << "4" << setw(20)<< x2*x2 << endl
<< setw(8) << "6" << setw(20)<< x3*x3 << endl;
getch();
return 0;
}
```

Output of the program

Number	Square
2	4
4	16
6	36

3.4. Operators in C++

Operators are the symbols which perform operation on operands. The operation may be arithmetic or comparison of data.

z=x+y;

Here, x, y and z are operands and '+' and '=' are the operators.

3.4.1. Operators in C++

C ++ is rich in operators and supports a wide range of these operators that range from the simplest unary to the complex ternary. Operators are very important because without their use, expressions cannot be evaluated.

The following are different types of operators used in C++.

a. Assignment = operator

The assignment operator assigns a value to a variable. The symbol = is used as an assignment operator. For example:

VarA = 55;

This statement assigns the integer value 55 to the variable VarA. The part at the left of the assignment operator = is a variable whereas the right side can be a constant value, a variable, the result of an operation or any combination of these.

The assignment operation always takes place from right to left.

VarA = VarB;

This statement assigns the value of variable VarB to VarA.

Consider the following program to demonstrate the use of assignment operator.

```
#include <iostream.h>
#include <conio.h>
int main()
{
int VarA, VarB;
VarA = 100;
VarB = 400;
cout<<" VarA :"<<VarA;
cout<<endl;
cout<<" VarB :"<<VarB;
getch();
```

```
return 0;
}
```

Output of the Program

VarA :100
VarB :400

b. Arithmetic operators

There are five arithmetic operators used in C++. These are (+, -, *, /, %) that support the following arithmetic operations shown in Table 3.4.

+	addition
−	subtraction
*	multiplication
/	division
%	modulus

Table 3.4: Arithmetic Operators

The percentage sign % shown in the above table is used for modulus. Modulus is the operation that gives the remainder of a division of two values. For example, if we write:

```
remainder = 11 % 5;
```

The variable 'remainder' will contain the value 1, since 1 is the remainder from dividing 11 by 5. The following program demonstrates the use of these operators.


```
#include <iostream.h>
#include <conio.h>
int main()
{
int a=11, b=5;
int sum=a+ b;
int difference=a - b;
int product=a * b;
float division=a / b;
int remainder=a % b;
cout<< "sum ="<<sum<<endl;
cout<< "difference ="<< difference <<endl;
cout<< "product ="<< product <<endl;
cout<< "division ="<< division <<endl;
cout<< "remainder ="<< remainder;
getch();
return 0;
}
```

Output of the Program

sum =16
difference =6
product =55
division =2.0
remainder =1

c. Compound or arithmetic assignment operators

Compound assignment operators used in C++ are: (+=, -=, *=, /=, %=). These operators are used to modify the value of a variable by performing an operation on the value that is

currently stored in that variable. For example, if we want to add the value of a variable y to the value of another variable x then the following statement is used.

x+=y;

i.e. x=x+y;

The list of these compound assignment operators along with their use is given in the Table 3.5.

Operator	Symbol	Form	Operation
Addition assignment	+=	x += y	Add y to x
Subtraction assignment	-=	x -= y	Subtract y from x
Multiplication assignment	*=	x *= y	Multiply x by y
Division assignment	/=	x /= y	Divide x by y
Modulus assignment	%=	x %= y	Put the remainder of x / y in x

Table 3.5: Arithmetic Assignment Operators

Consider the following example to demonstrate the use of compound assignment operators.

```
#include <iostream.h>
#include <conio.h>
int main()
{
int VarA, VarB=300;
VarA = VarB;
VarA +=200; // equivalent to VarA=VarA+200
cout<< "Value after addition="<<VarA;
getch();
return 0;
}
```


Output of the Program

Value after addition=500

d. Increment and decrement operators

These operators are also called unary arithmetic operators. These operators are used to make expressions short. The increment operator ++ and the decrement operator -- increase or decrease the value stored in a variable by 1. For example the following statements are equivalent and all perform the same operation of increasing the value of the variable 'VarC' by 1.

```
VarC ++;
VarC +=1;
VarC = VarC +1;
```

There are two versions of each of these increment and decrement operators. These are: a prefix version and a postfix version.

Prefix and Postfix forms

In **prefix** form of the increment and decrement operators, the operators precede their operands, e.g. ++x, --z

Consider the following segment of code:

```
// Prefix Increment and Decrement Operators
int x = 5;
cout<< "increment in prefix form ="<<++x;
cout<< "decrement in prefix form ="<<--x;
```

Output of the Segment

increment in prefix form =6

decrement in prefix form =4

In **postfix** form of the increment and decrement operators, the operands precede their operators, e.g. x++, z--

Consider the following program to demonstrate the use of increment and decrement operators.

```
// increment and Decrement Operators
#include<iostream.h>
#include<conio.h>
int main()
{
int x = 25, y = 25;
cout<< x << " " << y <<endl;
cout<< ++x << " " << --y <<endl; // prefix
cout<< x << " " << y <<endl;
cout<< x++ << " " << y-- <<endl; // postfix
cout<< x << " " << y <<endl;
getch();
return 0;
}
```

Output of the program

25 25

26 24

26 24

26 24

27 23

e. Relational operators

Relational operators are used for the comparison between two expressions. These operators are: ==, !=, >, >=, <, <=. The result of these operators can be either true or false. The list of all relational operators used in C++ is given in Table 3.8.

Operator	Symbol	Form	Operation
Greater than	>	x > y	true if x is greater than y, false otherwise
Less than	<	x < y	true if x is less than y, false otherwise
Greater than or equals	>=	x >= y	true if x is greater than or equal to y, false otherwise
Less than or equals	<=	x <= y	true if x is less than or equal to y, false otherwise
Equality	==	x == y	true if x equals y, false otherwise
Inequality	!=	x != y	true if x does not equal y, false otherwise

Table 3.8: List of Relational Operators

Consider the following lines of code:

(7 == 5) // false.

(5 > 4) // true.

(3 != 2) // true.

(6 >= 6) // true.

(5 < 5) // false.

f. Logical operators

There are three logical operators used in C++. These are: !, &&, || that are shown in Table 3.9 along with their operations.

Operator	Symbol	Form	Operation
Logical NOT	!	!x	true if x is false, or false if x is true
Logical AND	&&	x && y	true if both x and y are true, false otherwise
Logical OR	||	x || y	true if either x or y are true, false otherwise

Table 3.9: List of Logical Operators

i. Logical NOT

The ! operator is the C++ unary operator that is used to perform the Boolean NOT operation. The output of this operator is the inverse of the value of its operand. It produces **false** if its operand is **true** and **true** if its operand is **false**. The effects of logical NOT operator is summarized in the truth Table 3.10.

X	!X
True	False
False	True

Table 3.10: Logical NOT Operator

The following lines of code demonstrate the working of NOT operator.

```
!(5 == 5) // evaluates to false because the expression (5 == 5) is true.
!(6 <= 4) // evaluates to true because (6 <= 4) would be false.
!true// evaluates to false
!false// evaluates to true.
```

ii. Logical AND

Logical && operator is a binary operator that needs two operands. It is used to test whether both conditions are true or not. If both the conditions are **true**, the logical &&returns **true**, otherwise, it returns **false**. Truth Table 3.11 shows the results of logical &&operator.

X	Y	X && Y
False	False	False
False	True	False
True	False	False
True	True	True

Table 3.11: Logical AND Operator

iii. Logical OR

The logical || operator is used to test whether either of two conditions is true. If either of the left operand or right operand evaluates to **true**, the logical ||operator returns **true**. Also, if both the operands are **true**, logical ||returns to **true** but if both are **false**it returns to **false** Truth Table 3.12 shows the operations of logical || operator.

X	Y	X || Y
False	False	False
False	True	True
True	False	True
True	True	True

Table 3.12: Logical OR Operator

g. Ternary (or conditional) operator (? :)

The conditional operator takes three operands and is therefore also called ternary operator. It evaluates an expression and returns a value if that expression is true and a different value if the expression is false. The general syntax used for conditional operator is:

condition ? result1: result2;

Here, if condition is true the expression will return *result1*, and if condition is false it will return *result2*. Consider the following lines of code to demonstrate the use of conditional operator.

```
7==5 ?4 : 3; // returns 3, since 7 is not equal to 5.
7==5+2 ?4 : 3; // returns 4, since 7 is equal to 5+2.
5>3 ?a : b; // returns the value of a, since 5 is greater than 3.
```

The following program demonstrates the use of conditional operator.

```
// conditional operator
#include <iostream.h>
#include<conio.h>
int main()
{
int x, y, z;
x=22;
y=77;
z = (x>y) ? x : y;
cout<< "The greater value is: "<<z ;
getch();
return 0;
}
```

Output of the Program

The greater value is: 77

3.4.2. Unary, Binary and Ternary operators

The set of C++ operators is divided into three main categories: unary, binary and ternary. This division is on the basis of number of operands upon which the operators operate. The following sections discuss them.

a. Unary operators

Unary operators are those operators which need a single operand. There are two arithmetic unary operators: unary plus + and unary -.

Consider the following examples:

```
x = -y;
a = +b;
```


The logical ! operator is also a unary operator. Consider the following examples:

```
!x;
!(x > y);
```

Increment ++ and decrement -- operators are also the exampies of unary operators.

b. Binary operators

i. Binary operators are those operators which need two operands. For example (+, -, *, /, %)

ii. The assignment operator = is also a binary operator as it requires two operands. The operands are: the receiving variable on the left hand side and the evaluated expression on the right hand side.

iii. The logical and relational operators (<, <=, ==, >, >=, !=, &&, ||) are binary operators as well, as they require two operands.

c. Ternary operator

Ternary operator needs three operands. There is only one ternary operator. This operator is conditional operator (? :)

3.4.3. Expression

An expression consists of operators and operands or it may be the combination of variables and/or constants linked with operators. For example:

```
x=a + b;
Z= 2+ z / y;
y > 4;
```

The expressions are categorized as below:

i. **Arithmetic Expressions** having numeric operands and arithmetic operators and give numeric results. For example

```
x = y + 25;
sum = x + y;
```

ii. **Relational Expressions** having operands and relational operators for comparison of data. For example

```
x > y;
z >=10;
```

iii. **Logical Expressions** for testing of two or more Boolean expressions. For example

```
(x > y) || (x = 10);
(n=5) && (m > n);
```

3.4.4 Precedence of operators

Precedence of operator means that which operand will be evaluated first and which one will be evaluated later if the expression is a complex one. For example, in the following expression % will be evaluated first and then addition.

```
a = 5 + 7 % 2 /
```

From highest to lowest the priority is given below.

1. The parentheses are evaluated first.
2. Multiplication, division and modulus operators are performed next, from left to right at same level.
3. Addition and subtraction operators are resolved last, from left to right with same priority.

The following table shows the list of all operators with their precedence level.

Level	Operator	Description	Associativity
1	++ --	Postfix increment/decrement	left-to-right
2	++ -- + - !	Prefix increment/decrement Unary plus/minus Logical negation	right-to-left
3	* / %	Multiplication/division/modulus	left-to-right
4	+ -	Addition/subtraction	left-to-right
5	< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
6	== !=	Relational is equal to/is not equal to	left-to-right
7	&&	Logical AND	left-to-right
8	||	Logical OR	left-to-right
9	?:	Ternary conditional	right-to-left
10	= += -= *= /= %=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus assignment	right-to-left

Table 3.13 Operators Precedence

3.4.5 Compound expression

Compound expression is an expression that involves more than one sub-expressions linked with operators. Usually two or more relational expressions are linked with logical operators to form compound Boolean expression.

Following are the examples of compound expressions.

```
X1= (- B + sqrt( B*B – 4*A*C))/(2*A);
( (b * b) - 4 * a * c ) > 0
(age <= 20) && (height >= 5)
(m > 5) || (j > 0)
```

Summary

- C++ is a programming language developed by Bjarne Stroustrup which derives most of its features from C language.
- Each C++ program makes use of header files which have definitions for C++ objects.
- Pre-processor directives are the directives that are used to include header files to the C++ program.
- C++ Comments are the explanatory statements that are usually added to the source program statements to make them understandable to the readers.
- A C++ program needs variables to hold data upon which a programmer performs operations.
- Constant are those components of C++ language whose values cannot be changed during the execution of a program.
- Data types are the qualifiers that tell the compiler what type of data will be stored in a particular variable.
- For the declaration of variables, one needs to select proper data type.
- C++ supports a wide range of data types, such as, int, float, double char etc.

- For getting input and producing output, C++ has a rich library of pre-defined and pre-compiled objects, cin and cout, and functions such as getch (), getchar (), gets, putch () and putchar ().
- Escape sequences are special characters that start with back slash symbol ('\'). The subsequent character after ('\') symbol has special meaning.
- Operators are those symbols which operate over the operands.
- C++ supports a wide range of operators to perform operations on operands.
- Arithmetic operators are those operators that are used in arithmetic expressions and perform arithmetic operations.
- Relational operators are used for comparing two values and return (1) if the condition is TRUE and (0) if FALSE.
- Unary operators take one operand; binary operators take two operands and ternary operator takes there operands.

Exercise

Q.1 Fill in the Blanks.

i. In C/C++ language, a hexadecimal number is represented by writing _____ symbol.
ii. (?) is the _____ operator used in C++.
iii. Output of the logical and relational operators is _____.
iv. Assigning values to a variable during declaration time is called _____.
v. Each C++ program has _____ function from where the execution of the program starts.

Q.2 Select the correct choice for the following Multiple Choice Questions.

i. What does the given expression evaluate to? E=6 + 5 * 4 % 3
   a. 1
   b. 8
   c. 11
   d. None of the above
ii. (true && true) || false
   a. true
   b. false
   c. both a and b
   d. none of the above
iii. Expression C=i++ causes
   a. Value of i assigned to C and then i incremented by 1
   b. to be incremented by 1 and then value of i assigned to C
   c. Value of i assigned to C
   d. to be incremented by 1

iv. Which of the following languages is a subset of C++ language?
   a. C language
   b. Java language
   c. B language
   d. C# language
v. In C++ language char ch='3' represents:
   a. A digit
   b. An integer constant
   c. A character constant
   d. A word

Q.3 Write TRUE/FALSE against the following statements.

i. Definition for the C++ object cout is stored in header file conio.h.
ii. In C++ a statement ends with ':' symbol.
iii. *int* **20sum** is the correct declaration of the variable **20sum**.
iv. In C++ the statement a=a+20 is the same as a+=20.
v. Size of *short int* data type is 4 bytes.

Q.4 Evaluate the following:

1) (false && true) || true
2) (false && true) || false || true
3) (5 > 6 || 4 > 3) && (7 > 8)
4) !(7 > 6 || 3 > 4)

Q.5 Pick those variables from the following list which are improperly declared and named. Also, describe the reason of its invalidity?

1) int sum;
2) int cats=5, dogs=5;