UNIT I

Introduction to Object Technology - Objects and Methods - Classes - Declaring classes and objects - Concept of OOAD - Key Object Orientation Concepts and Elementary C++ programming - I/O statements - Fundamental types - Operators.

INTRODUCTION

The C++ programming language was developed at AT&T Bell Laboratories by Bjarne Stroustrup. He found that C was lacking with certain functionalities and decided to extend the language by adding features from his favorite language, Simula 67. Simula 67 was one of the earliest object-oriented languages. He called the new language as "C with Classes" and later renamed as "C++" (++ is the increment operator).

C++ CHARACTER SET

A character set is the valid set of characters that a language can recognize. A character can be a letter, digit or any other sign.

Letters	A-Z, a-z
Digits	0-9
Special Symbols	+ - * / ^ ( ) [ ] = !=< > . , ' " $: ; % ! & ? { } _ @ \ |
White spaces	Blank space, new line, horizontal tab

C++ TOKENS

The smallest individual unit in a program is called as token. The C++ tokens are keywords, identifiers (variables), literals, punctuators and operators.

1. Keywords: The keywords carry a special meaning for the C++ compiler.

Eg: auto, break, case, char, class, const, continue, default, delete, do, else, enum, false, float, for, friend, goto, if, long, new, short, return, private, public, protected, sizeof, static, struct, switch, this, true, throw, try, union, virtual, void, while etc.

2. Identifiers: An identifier is the name given by user for a unit of program.

The rules to form an identifier are:

2.1 Combination of letters and digits
2.2 First character should be a letter
2.3 No special character is allowed except underscore
2.4 Upper and lower case letters are significant
2.5 No keyword should be used as an identifier

Eg:

Valid identifiers	Name File07 avg_age
Invalid identifiers	break hai-name file.19 my,name

Two types of identifiers are variables and constants. A value stored at a variable can be changed at time within the program while the value stored at a constant remains unchanged throughout the program.

3. Literal: Literals are the data items that never change their value during a program run. There are four kinds of literal as:
   3.1 Integer constants are whole numbers without any fractional part. C++ allows three types of integer constants as,
      a. **Decimal Integer Constant** consists of a sequence of digits with or without sign. Eg: 1234, -98, +45
      b. **Octal Integer Constant** is the sequence of digits starting with 0(digit zero). Eg: Decimal 8 is written as 010, Decimal 12 is written as 014. Make sure the octal integer constant should contain digits from 0 to 7.
      c. **Hexadecimal Integer Constant** is the sequence of digits starting with 0X or 0x (digit zero). Eg: Decimal 12 is written as 0XC.
   3.2 Character constant is single character enclosed in single quotes. Eg: 'C'. An escape sequence represents a single character. Eg: \a, \b, \n, \t, \v, \0 etc.
   3.3 Floating constant or Real constant is the numbers with fractional part. They can be written in two formats:
      a. Fractional form consists of signed or unsigned digits including a decimal point between digits. It must have one digit before and after decimal point. A real constant with no sign is assumed to be positive. Eg:

Valid	2.0, -17.5
Invalid	7, 8., +17/2,15.67.2

      b. Exponent form consists of two parts: mantissa and exponent. A mantissa must be either integer or real with or without sign. The mantissa is followed by a letter E or e. The exponent should be integer.

Valid	15E09, 0.152e4, 13e+3 14.5e-2
Invalid	1.7e, .25e-7, 17e2.3

   3.4 String literals are characters enclosed in double quotes. Each string literal is automatically added by \0 at the end by compiler itself.
4. Punctuators: The following characters are used as punctuators or separators. [ ] { } ( ) : ; ' " .
5. Operators: They are tokens that trigger some computation or action when applied to variables and other objects in an expression. Different types of operators are arithmetic operators, logical operators, bitwise operators, conditional operators and special operators.

C++ DATA TYPES

Data type is means to identify the type of data and associated operations for handling it. The different types of data are:

Fundamental data type	int float char double void
Derived data type	Array pointer reference functions
User-defined data type	class structure union enumeration

Fundamental (atomic) data types are those that are not composed of other data types.
Eg:

1. int data type: Integers are whole numbers with no fractional part. Integers are represented by keyword 'int'. An identifier declared as int will store only integer numbers. It occupies 2 bytes memory space.
2. char data type: It can store any character from C++ character set. Char type is often said to be an integer type because they are represented in memory by associated number codes (ASCII). It is specified by the keyword 'char'. It occupies 1 byte memory space.
3. float data type: It is the number with decimal point. It can be represented as fractional part or scientific notation. It is specified by the keyword 'float'. It occupies 4 bytes memory space.
4. Double data type: It is used to handle floating point numbers. It occupies twice memory space as that of float and so it is higher in precision. It is specified by the keyword 'double'. It occupies 8 bytes memory space.
5. void data type:It specifies an empty set of values. It is used as the return type of the function that returns no value. It is specified by the keyword 'void'. It occupies 1 byte memory space.

OPERATORS

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators. Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.

1. **Assignment (=) operator:** The assignment operator assigns a value to a variable. Eg: a = 5;

This statement assigns the integer value 5 to the variable a. The part at the left of the assignment operator (=) is known as the lvalue (left value) and the right one as the rvalue (right value). The lvalue has to be a variable whereas the rvalue can be a constant, a variable, the result of an operation or any combination of these. The

most important rule when assigning is the right-to-left rule: The assignment operation always takes place from right to left, and never the other way: a = b;
This statement assigns to variable a (the lvalue) the value contained in variable b (the rvalue). The value that was stored until this moment in a is not considered at all in this operation, and in fact that value is lost. Consider also that we are only assigning the value of b to a at the moment of the assignment operation. Therefore a later change of b will not affect the new value of a.
This code will give us as result that the value contained in a is 4 and the one contained in b is 7. Notice how a was not affected by the final modification of b, even though we declared a = b earlier (that is because of the right-to-left rule). A property that C++ has over other programming languages is that the assignment operation can be used as the rvalue (or part of an rvalue) for another assignment operation.
For example:
a = 2 + (b = 5);
is equivalent to: b = 5;a = 2 + b;
that means: first assign 5 to variable b and then assign to a the value 2 plus the result of the previous assignment of b (i.e. 5), leaving a with a final value of 7. The following expression is also valid in C++: a = b = c = 5; It assigns 5 to the all the three variables: a, b and c.

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

The five arithmetical operations supported by the C++ language are:
+ addition
- subtraction
* multiplication
/ division
%modulo
Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that we might not be so used to see may be modulo; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:
a = 11 % 3;
the variable a will contain the value 2, since 2 is the remainder from dividing 11 between 3.

3. **Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=):**

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); and the same for all other operators.

4. **Increment and decrement (++, --):**

Shortening even more some expressions, the increment operator (++) and the decrement operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

```
c++;
c+=1;
c=c+1;
```

are all equivalent in its functionality: the three of them increase by one the value of c. A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (++a) or after it (a++). Although in simple expressions like a++ or ++a both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (a++) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression.

Notice the difference:

Example 1

```
 B=3;
A=++B;
// A contains 4, B contains 4
```

Example 2

```
B=3;
A=B++;
// A contains 3, B contains 4
```

In Example 1, B is increased before its value is copied to A. While in Example 2, the value of B is copied to A and then B is increased.

5. **Relational and equality operators (==, !=, >, <, >=, <=):**

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result. We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is. Here is a list of the relational and equality operators that can be used in C++:

==Equal to
!= Not equal to
> Greater than
< Less than
>=Greater than or equal to

<=Less than or equal to
Here there are some examples:
(7 == 5) // evaluates to false.
(5 > 4) // evaluates to true.
(3 != 2) // evaluates to true.
(6 >= 6) // evaluates to true.
(5 < 5) // evaluates to false.
Of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that a=2, b=3 and c=6,
(a == 5) // evaluates to false since a is not equal to 5.
(a*b >= c) // evaluates to true since (2*3 >= 6) is true.
(b+4 > a*c) // evaluates to false since (3+4 > 2*6) is false.
((b=2) == a) // evaluates to true.

The operator = (one equal sign) is not the same as the operator == (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one (==) is the equality operator that compares whether both expressions in the two sides of it are equal to each other. Thus, in the last expression ((b=2) == a), we first assigned the value 2 to b and then we compared it to a, that also stores the value 2, so the result of the operation is true.

6. Logical operators (!, &&, ||):

The Operator! is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand.
Eg:
!(5 == 5) // evaluates to false because the expression at its right (5 == 5) is true.
!(6 <= 4) // evaluates to true because (6 <= 4) would be false.
!true // evaluates to false
!false // evaluates to true.

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator && evaluating the expression a && b:

a	b	a&&b
T	T	T
T	F	F
F	T	F
F	F	F

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of a || b:

a	b	a||b
T	T	T
T	F	T
F	T	T
F	F	F

((5 == 5) && (3 > 6)) // evaluates to false (true && false).
((5 == 5) || (3 > 6)) // evaluates to true (true || false).

7. Conditional operator (?):

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:

condition ? result1 : result2

If condition is true the expression will return result1, if it is not it will return result2.
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.
a>b ? a : b // returns whichever is greater, a or b.
In this example a was 2 and b was 7, so the expression being evaluated (a>b) was not true, thus the first value specified after the question mark was discarded in favor of the second value (the one after the colon) which was b, with a value of 7.

8. Comma operator (,):

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.
Eg:
a = (b=3, b+2);
Would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

9. Bitwise Operators (&, |, ^, ~, <<, >>):

Bitwise operators modify variables considering the bit patterns that represent the values they store.

Operator	Meaning	Description
&	AND	Bitwise AND
|	OR	Bitwise OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary Complement
<<	SHL	Shift Left
>>	SHR	Shift Right

Precedence of operators

When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:
a = 5 + 7 % 2
We may doubt if it really means:
a = 5 + (7 % 2) // with a result of 6, or
a = (5 + 7) % 2 // with a result of 0
The correct answer is the first of the two expressions, with a result of 6. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference come from mathematics) but for all the operators which can appear in C++. From greatest to lowest priority, the priority order is as follows:

Level	Operator	Description	Associativity
1	::	scope qualifier	Left-to-right
2	++ --	postfix increment / decrement	Left-to-right
	()	functional forms	
	[]	subscript	
	. ->	member access	
3	++ --	prefix increment / decrement	Right-to-left
	~ !	bitwise NOT / logical NOT	
	+ -	unary prefix	
	& *	reference / dereference	
	new delete	allocation / deallocation	
	sizeof	parameter pack	
	(*type*)	C-style type-casting	
4	.* ->*	access pointer	Left-to-right
5	* / %	multiply, divide, modulo	Left-to-right
6	+ -	addition, subtraction	Left-to-right
7	<< >>	shift left, shift right	Left-to-right
8	< > <= >=	comparison operators	Left-to-right
9	== !=	equality / inequality	Left-to-right
10	&	bitwise AND	Left-to-right
11	^	bitwise XOR	Left-to-right
12	|	bitwise OR	Left-to-right
13	&&	logical AND	Left-to-right

14	||	logical OR	Left-to-right
15	= *= /= %= += -= >> = <<= &= ^= |=	assignment / compound assignment	Right-to-left
	?:	conditional operator	
16	,	comma separator	Left-to-right

Grouping/Associativity defines the precedence order in which operators are evaluated in the case that there are several operators of the same level in an expression. All these precedence levels for operators can be manipulated or become more legible by removing possible ambiguities using parentheses signs (and), as in this example:

a = 5 + 7 % 2;

might be written either as: a = 5 + (7 % 2); or a = (5 + 7) % 2; depending on the operation that we want to perform. Note: So if we want to write complicated expressions and we are not completely sure of the precedence levels, always include parentheses. It will also become a code easier to read.

EXPRESSIONS

An expression is any computation which yields a value. An expression evaluates to a certain value. These are permanent changes in the program state. In this sense, C++ expressions are different from mathematical expressions. Expressions may be of the following seven types:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

An expression may also use combinations of the above expressions. Such expressions are known as compound expressions

1. **Constant Expressions** consist of only constant values. Examples:

15

20 + 5 / 2.0

2. **Integral Expressions** are those which produce integer results after implementing all the automatic and explicit type conversions. Examples:

m + n

m * n – 5

5 + int(2.3)

where m and n are integer variables.

3. **Float Expressions** are those which, after all conversions, produce floating

– point results. Examples:

x * y / 10
5 + float(10)
where x and y are floating – point variables.

4. **Pointer Expressions** produce address values. Examples:

&m
ptr + 1
where m is a variable and ptr is a pointer.

5. **Relational Expressions** yield results of type *bool* which takes a value *true* or *false*. Examples:

x <= y
a+b == c+d
When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as ***Boolean expressions.***

6. **Logical Expressions** combine two or more relational expressions and produce *bool* type results. Examples:

a>b && x==10 x==10 || y==5

7. **Bitwise Expressions** are used to manipulate data at a bit level. They are basically used for testing or shifting bits. Examples:

```
x << 3// Shift three bit position to left
y >> 1//Shift one bit position to right
```

Shift operators are often used for multiplication and division by powers of two.

TYPE CASTING

Converting an expression of a given type into another type is known as type-casting. We have already seen some ways to type cast:

Implicit conversion: Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type.

Eg:

```
short a=2000;
int b;
b=a;
```

Here, the value of a has been promoted from short to int and we have not had to specify any type-casting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This can be avoided with an explicit conversion.

Explicit type casting operator

Type casting operators allow you to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited

from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int ( f );
```

Both ways of type casting are valid in C++.

sizeof()

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

a = sizeof (char); This will assign the value 1 to a because char is a one-byte long type. The value returned by sizeof is a constant, so it is always determined before program execution.

DECLARATION OF VARIABLES

Every variable in C++ is declared as follows:

[specifier] datatype name1 [=value]; where

specifier=optional keyword like const, static
datatype=any valid data type allowed in C++
name=valid identifier name
value=It is optional and used if we want to initialize any value.

The purpose of declaration is to introduce name to the program, (ie) to explain the compiler what the name means. The data type tells the compiler what range of value can be at the value and what operations can be performed. It is customary to use capital letters for constant identifiers and small letters for variable identifiers. The compiler will replace each constant symbol with its numeric value. The location of declaration determines the scope of the variable-the part of the program where the variable may be used. The word 'variable' is used to suggest that the object's value can be changed. An object whose value cannot be changed is called as a constant. Constants are declared by preceding its type specifier 'const'. Constants must be initialized at the place of declaration itself. It should be noted that more than one variable can be declared in a single declaration statement delimited by comma as

datatype name1,name2,...namen;

Eg:

```
const int MAX=10;
```

int x,y,z; (or) int x; int y; int z;
float a,b;

So an object is a contiguous region of memory with an address (memory address of first byte), a size (number of bytes occupies in memory), a type (data type) and a value (actual value stored in it).

I/O STATEMENTS

Output Statement:

By default, the standard output of a program is the screen, and the C++ stream object defined to access it is ***cout***. ***cout*** is used in conjunction with the ***insertion operator***, which is written as << (two "less than" signs).

Syntax: **cout << "Output sentence";**//prints Output sentence on screen

```
cout << 120; // prints number 120 on screen
cout << x;// prints the content of x on screen
```

The << operator inserts the data that follows it into the stream preceding it. In the examples above, it inserted the constant string Output sentence, the numerical constant 120 and variable x into the standard output stream cout. Notice that the sentence in the first instruction is enclosed between double quotes (") because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly distinguished from variable names. For example, these two sentences have very different results:

```
cout << "Hello"; // prints Hello
cout << Hello;//prints the content of Hello variable
```

The insertion operator (<<) may be used more than once in a single statement. The utility of repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

```
cout << "Hello, I am " << age << " years old and my postal code is " << pincode;
```

If we assume the age variable to contain the value 24 and the pincode variable to contain 400054 the output of the previous statement would be:

Hello, I am 24 years old and my postal code is 400054

It is important to notice that ***cout*** does not add a line break after its output unless we explicitly indicate it, therefore, the following statements:

```
cout << "This is a sentence.";
cout << "This is another sentence.";
```

will be shown on the screen one following the other without any line break between them:

This is a sentence. This is another sentence. (even though we had written them in two different insertions into cout). In order to perform a line break on the output we must explicitly insert a new-line character into cout. In C++ a new-line

character can be specified as ***\n (backslash, n)***:
cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";
This produces the following output:
First sentence.
Second sentence.
Third sentence.
Additionally, to add a new-line, you may also use the ***endl*** manipulator. For example:
cout << "First sentence." << endl; (This is also called as cascading of operator).
cout << "Second sentence." endl;
would print out:
First sentence. Second sentence.
The ***endl*** manipulator produces a newline character, exactly as the insertion of '***\n***' does, but it also has an additional behaviour when it is used with buffered streams: the buffer is flushed. Anyway, cout will be an unbuffered stream in most cases, so you can generally use both the ***\n*** escape character and the ***endl*** manipulator in order to specify a new line without any difference in its behavior.

Input Statement:
The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream.
Syntax:
cin>>variable name1>>variable name2>>...variable namen;
For example: int age; cin >> age;
The first statement declares a variable of type int called age, and the second one waits for an input from cin (the keyboard) in order to store it in this integer variable. cin can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if we request a single character, the extraction from cin will not process the input until the user presses RETURN after the character has been introduced.
We must always consider the type of the variable that we are using as a container with cin extractions. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.
cin >> a >> b; is equivalent to:
cin >> a; cin >> b;
In both cases the user must give two data, one for variable a and another one for variable b that may be separated by any valid blank separator: a space, a tab character or a newline.

cin and Strings: We can use cin to get strings with the extraction operator (>>) as we do with fundamental data type variables:
cin >> mystring;
However, as it has been said, cin extraction stops reading as soon as if finds any blank space character, so in this case we will be able to get just one word for each extraction.

GENERAL STRUCTURE OF A C++ PROGRAM

As C++ is a programming language so it follows a predefined structure. The program is divided into many sections, it is important to know the need of every section. The easiest way to understand the basic structure of C++ program is by writing a program. The basic C++ program is as follows:

```
Documentation Section
Preprocessor Directives
using namespace std;
Global Variable Declaration
Class Definitions
int main()
{
Local variable declaration
Executable Statements
return 0;
}
```

- **Documentation Section:** This section comprises of comments. As the name suggests, this section is used to improve the readability and understanding of the program.// (Double Slash) represents comments in C++ program. Comments can be of single line or multiple lines. Double Slash comments are used to represent single line comments. For multiple line comment, you can begin with /* and end with */.

For example :

```
/*  Text line number 1
Text line number 2
Text line number 3 */
```

- **Linking and Directives Section :** A program begins with #include<iostream>. <iostream> represents header file which includes the functionalities of predefined functions. In linking section, the compiler in-built functions such as cout<<, cin>> etc are linked with INCLUDE subdirectory's header file <iostream>. The '#' symbol tells about "address to" or "link to". iostream is input/output stream which includes declarations of standard input-output library in C++.
- **main() Section :** This is the section in which the program coding is written. Basically, it acts as a container for C++ program. The execution of the C++

program begins with main() function and it is independent of the location of main() function in the program. main() is a function and so it carries parenthesis "()". The body of the main() function can be found right after these parenthesis, the body is enclosed in braces "{ }".

- **Body of main() Section :** The body of the main() function begins with "{".
  - ✓ Local Variable Declaration : In this the variables which are used in the body of the main() functions are declared. These are called the local variables as their scope is limited within the main() function only, unless they are declared globally outside the main() function.
  - ✓ Statements to Execute : This section includes statements for reading, writing and executing data using I/O functions, library functions, formulas, conditional statements etc. It should be noted that every statement should be terminated by ";" (semicolon). The statements are written in separate lines just to improve the readability of the program. C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where on a line we put a statement. For example:

```
x = y;
y = y+1;
add(x, y);
```

is the same as

```
x = y; y = y+1; add(x, y);
```

- ✓ **return 0;** in the above program causes the function to finish and **0** represents that function has been executed with zero errors. This is considered as most usual way to end a C++ program.
- ✓ Finally the body of the main() function ends with "}".

Eg:

```
#include <iostream>
using namespace std;
// main() is where program execution begins.
int main()
{
  cout << "Hello World"; // prints Hello World
  return 0;
}
```

Let us look various parts of the above program:

- The C++ language defines several headers, which contain information that is either necessary or useful to the program. For this program, the header **<iostream>** is needed.

- The line **using namespace std;** tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.
- The next line **// main() is where program execution begins.** is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
- The line **int main()** is the main function where program execution begins.
- The next line **cout << "This is my first C++ program.";** causes the message "This is my first C++ program" to be displayed on the screen.
- The next line **return 0;** terminates main( )function and causes it to return the value 0 to the calling process.

Compile & Execute C++ Program:

- Open a text editor and add the code as above and save the file as: hello.cpp
- Open a command prompt and go to the directory where you saved the file.
- Type 'g++ hello.cpp ' and press enter to compile the code. If errors, go and correct it in the program.
- If there are no errors in the code type ' a.out' to run the program.
- We will be able to see ' Hello World ' printed on the window.

```
$ g++ hello.cpp
$ ./a.out
Hello World
```

Make sure that g++ is in your path and that you are running it in the directory containing file hello.cpp.

```
x = y;
y = y+1;
add(x, y);
```

A block is a set of logically connected statements that are surrounded by opening and closing braces. For example:

```
{
  cout << "Hello World"; // prints Hello World
  return 0;}
```

Note: A recent addition in C++ program is **using namespace std;** after preprocessor directives. cout is part of the standard library, and all the elements in the standard C++ library are declared within what is a called a *namespace*: the namespace std. In order to refer to the elements in the std namespace a program shall either qualify each and every use of elements of the library (as we have done by prefixing cout with std::), or introduce visibility of its components.

INTRODUCTION TO OBJECT-ORIENTED TECHNOLOGY

Structured Programming was a powerful tool to write moderately complex programs fairly easy-not for large programs since it is lack in bug-free, easy-to-maintain, reusable programs. OOP is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts".

Procedure-Oriented Programming (POP): A number of functions are implemented and the problem is viewed as a sequence of things like reading, calculating and printing. **Eg:** COBOL, ForTran, C

POP has a list of instructions for the computer to follow and organize into groups called functions. Flowchart is used to represent the flow of control. What about data? Many important data items are placed as global so that all functions access them. Each function has its own location data. In a large program, it is hard to keep track of all data and also it does not represent real-world entities.

1. Emphasis on doing things
2. Large programs are divided into functions
3. Functions share global data
4. Data move openly around the system from function to function
5. Function transform data from one form to other
6. Top-Down approach

Object-Oriented Programming (OOP): OOP concentrates on data and does not allow the free movement of data. It ties the data more closely to the functions using the data and protects from accidental modification. The fundamental principle of object

technology is Object (everything we touch, feel and smell). The origin of the object technology can be at 1960s by PL called Simula (Simulation language used to simulate complex and huge physical systems). In Simula, models were created as objects. **Eg:** C++, Java.

An **Object** is a self-contained entity (i.e.) it encloses both data and operations. In procedural code, the emphasis is on operations and data comes at end. In object technology, we are considering operations and data as two portions of a same object—data can be no longer separated and it is tightly coupled with the operations. Therefore program is simply a collection of well-designed objects.
Eg: In a Payroll application, employee, department, manager, salary are objects.

Terminology:

2) In object technology, the operations are called **methods or member functions.**
3) Data items or variables are called **variables or data members**.

We can pass messages to an object to do operations. Since every object must list out the methods it supports. This enables other objects to consult this list and send it appropriate messages. Objects can call the methods of other object by sending messages to each other. However, an object does not have any direct access to the data portion of another object. An object hides its data from the outside world. But is publishes its methods to the entire world.

1. Emphasis on data rather than on procedure
2. Programs are divided into objects
3. Data structures are designed such that they represent objects
4. Object ties up data and functions together

5. Data is not accessible to the external functions
6. Objects communication through functions
7. New data and functions can be added wherever necessary
8. Bottom-Up approach

BASIC CONCEPTS OF OOPS

1. **Objects:** Basic runtime entities in an Object-Oriented system are called as Objects similar to record or structure in C and Pascal. It may represent a person, place, bank account, table of data etc.

2. **Class:** The entire set of data and code of an object can be made by a user-defined data type called as class. Objects are the variables of class. Class is the collection of objects of similar type. We can create many objects for a class. Class is a user-defined data type but behave like a built-in type. **Eg**: vegetable tomato; fruit apple; flower rose;
3. **Data Abstraction and Encapsulation:** Wrapping of data and function within a class is called as encapsulation (most striking feature of a class). Data which is not accessible to the outside world referred as **data hiding**. Act of representing essential features without including the background details or explanations is called as **abstraction**. Class is called as **ADT (Abstract Data Type).**
4. **Inheritance:** It refers to the concept in which one object acquires the properties of other object. It uses the idea of reusability. Addition of the additional features to an existing class without modifying it is possible.

5. **Polymorphism:** It is a Greek term refers to the ability to take more than one form. An operation many exhibit different behaviors in different instances-behavior depends upon the types of data. **Eg**: "+" means addition of two numbers and also concatenation of two strings. Process of making an operator to exhibit different behaviors in different instances is called as operator overloading. A word has different meanings depending upon the context. Similarly, a function can be used to handle different number and type of arguments is called as function overloading.
6. **Dynamic Binding:** Linking of procedure call to the code to the executed is called as binding. Dynamic Binding or Late Binding means that the code associated with the given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance.

7. **Message Passing:** Objects communicate with each other like people communicate through messages. Message is a request for execution of a procedure. Message passing involve specifying the name of the object, function name and information to be sent. Objects have life (i.e.) they can be created and destroyed. Therefore communication can be done if and only if object is alive.

Eg: employee.salary(name);

OOAD

OOAD means **Object Oriented Analysis and Design** where SSAD means **Structured System Analysis and Design**. OOAD becomes very popular and it has become the standard for many software projects. Both techniques involve a similar set of steps like analysis, design, development, testing and implementation; but OOAD differs of what is done in these phases. Grady Booch, Evor Jacobson and James Rambough have given major contribution to OOAD methodologies.

BENEFITS OF OOPS

- We can eliminate redundant code and extend the use of existing classes through inheritance.

- We can build programs from the standard working modules, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- Data hiding helps to build secure programs
- Possible to have multiple instances of a class that can exist without any interference
- Possible to map objects in the problem domain to those in the program
- Easy to partition the project into object
- The data-centered design approach enables us to capture more details of a model
- Object-oriented system can be easily upgraded from small to large
- Message-passing techniques make interface simpler
- Software complexity can be easily managed or reduced
- Allows maintability and reusability

APPLICATIONS OF OOP

Real-time systems, Simulation and modeling, Object-oriented databases, Hypertext and hypermedia, AI and expert system, Neural networks and parallel programming, Decision support and office-automation system
CIM, CAM, CAD system, User-interface design

UNIT II

Control Structures: Decision making statements- if, if-else, else-if ladder, nested if, Switch statements - Looping Statements- while, do-while, for - break, continue, goto statements.

CONTROL STRUCTURES

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances. With the introduction of control structures we are going to have to introduce a new concept: the compound-statement or block. A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: { }:. There are 3 types of statements as sequence, selection and iteration.

SELECTION STATEMENTS

There are three types of decision making statements in C++. They are:

- if statements

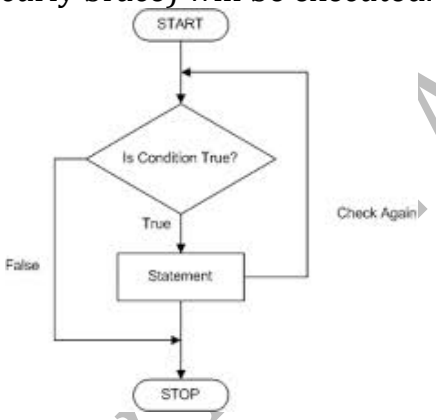
- switch statements
- jump statements

1.1 The if Statement: An if statement consists of a Boolean expression followed by one or more statements.

Syntax: The syntax of an if statement is:

```
if(Boolean_expression)
{
  //Statements will execute if the Boolean expression is true
}
```

If the boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement (after the closing curly brace) will be executed.



Example:

```
#include<iostream>
using namespace std;
int main()
{
    int x = 10;
    if( x < 20 )
       cout<<"This is if statement";
   return 0;
}
```

This would produce following result:

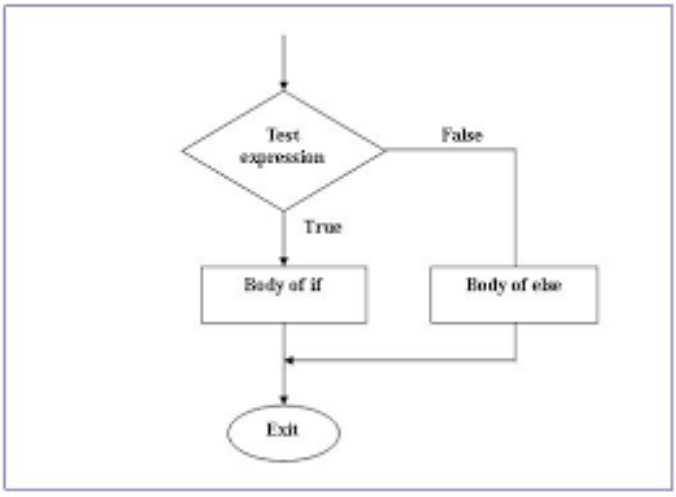
```
This is if statement
```

1.2 The if...else Statement: An if statement can be followed by an optional *else* statement, which executes when the Boolean expression is false.

Syntax: The syntax of if..else block is,

```
if(Boolean_expression)
{
  //Executes when the Boolean expression is true
```

```
}
else
{
  //Executes when the Boolean expression is false
}
```



Example:

```
#include<iostream>
using namespace std;
int main()
{
    int x = 30;
    if( x < 20 )
      cout<<"This is if statement";
    else
      cout<<"This is else statement";
   return 0;
 }
```

This would produce following result:

```
This is else statement
```

1.3 The Else if ladder Statement: An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement. When using if, else if, else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax: The syntax of a if...else is:

```
if(Boolean_expression 1)
{
  //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2)
{
  //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3)
{
  //Executes when the Boolean expression 3 is true
}else
{
  //Executes when none of the above condition is true.
}
```

Example:

```
#include<iostream>
using namespace std;
int main()
{
    int x = 30;
    if( x == 10 )
      cout<<"Value of X is 10";
    else if( x == 20 )
      cout<<"Value of X is 20";
    else if( x == 30 )
      cout<<"Value of X is 30";
    else
      cout<<"This is else statement";
    return 0;
}
```

This would produce following result:

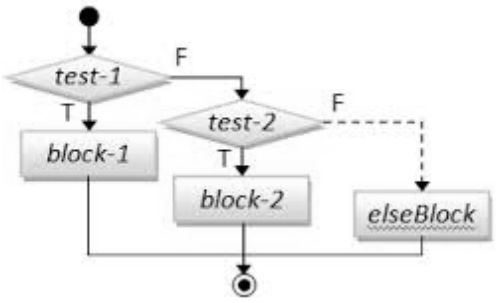
```
Value of X is 30
```

1.4 Nested if Statement: It is always legal to nest if-else statements, which means we can use one if or else if statement inside another if or else if statement.

Syntax: The syntax for a nested if...else is as follows:

```
if(Boolean_expression 1)
{
  //Executes when the Boolean expression 1 is true
  if(Boolean_expression 2)
  {
    //Executes when the Boolean expression 2 is true
  }
}
```

We can nest *else if...else* in the similar way as we have nested *if* statement.



Example:

```
#include<iostream>
using namespace std;
int main()
{
    int x = 30;   int y = 10;
    if( x == 30 )
   {
      if( y == 10 )
        cout<<"X = 30 and Y = 10";
   }
      return 0;
}
```

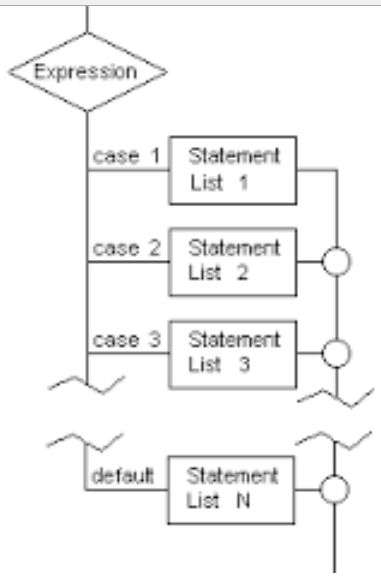
This would produce following result:

```
X = 30 and Y = 10
```

1.5 The switch Statement: A *switch* statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax: The syntax of enhanced for loop is:

```
switch(expression)
{
   case value :
     //Statements
     break;
   case value :
     //Statements
     break;
   //We can have any number of case statements.
   default : //Optional
     //Statements
}
```



The following rules apply to a switch statement:

- The variable used in a switch statement can only be a byte, short, int, or char.
- We can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.

- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
#include<iostream>
using namespace std;
int main()
{
    char grade = 'C';
    switch(grade)
    {
    case 'A' :
      cout<<"Excellent!";
      break;
    case 'B' :
    case 'C' :
      cout<<"Well done";
      break;
    case 'D' :
      cout<<"We passed";
    case 'F' :
      cout<<"Better try again";
      break;
    default :
      cout<<"Invalid grade";
    }
    System.out.println("Grade is " + grade);
    return 0;
}
```

This would produce following result:

```
Well done
Grade is a C
```

ITERATIVE STATEMENTS

Loop or iteration refers to the process of repeating a statement or a group of statements repeatedly. The loops can be of two types as entry controlled loop and exit controlled loop. The entry controlled loops will first check the condition and

then execute the block depending upon the status of condition. The exit controlled loop will execute the block and then check for the condition. C++ has very flexible three looping mechanisms. We can use one of the following three loops:

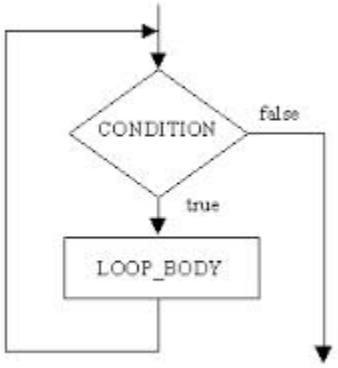
- while Loop
- do...while Loop
- for Loop

2.1 The while Loop: A while loop is a control structure that allows repeating a task a certain number of times.

Syntax: The syntax of a while loop is:

```
while(Boolean_expression)
{
  //Statements
}
```

If the *boolean_expression* result is true then the actions inside the loop will be executed. This will continue as long as the expression result is true. Here key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed. Hence while loop is an entry-controlled loop.



Example:

```
#include<iostream>
using namespace std;
int main()
{   int x= 10;
    while( x < 20 )
    {
      cout<<"value of x : " + x ;
      x++;
      cout<<"\n";
    }
    return 0;
```

```
}
```

This would produce following result:

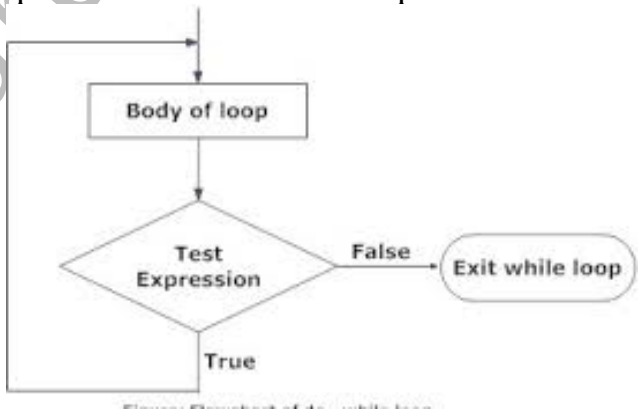
```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

2.2 The do...while Loop: A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time. Hence do..while loop is an exit-controlled loop.

Syntax: The syntax of a do...while loop is:

```
do
{
  //Statements
}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested. If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.



Example:

```
#include<iostream>
using namespace std;
```

```
int main()
{
    int x= 10;
    do
   {
      cout<<"value of x : " + x ;
      x++;
      cout<<"\n";
    }
  while( x < 20 );
return 0;
  }
```

This would produce following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

2.3 The for Loop: A for loop is a repetition control structure that allows us to efficiently write a loop that needs to execute a specific number of times. A for loop is useful when we know how many times a task is to be repeated. For loop is also an entry controlled loop.

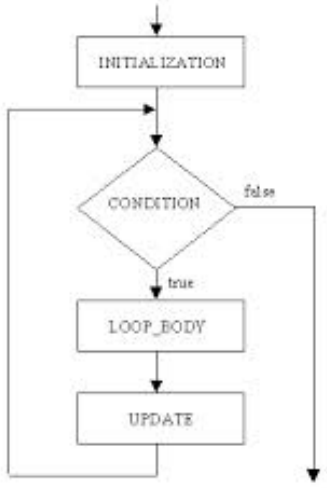
Syntax: The syntax of for loop is:

```
for(initialization; Boolean_expression; update)
{  //Statements
}
```

Here is the flow of control in for loop:

1. The initialization step is executed first, and only once. This step allows we to declare and initialize any loop control variables. We are not required to put a statement here, as long as a semicolon appears.

2. Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past for loop.
3. After the body of for loop executes the flow of control jumps back up to the update statement. This statement allows us to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
4. The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, for loop terminates.
5. A for loop may contain multiple initializations or multiple updates each separated by comma.
6. Under certain circumstances any part of for loop can be omitted.
7. for(;;) will produce infinite loop.
8. Nested for loops are allowed.



Example:

```
#include<iostream>
using namespace std;
int main()
{
    for(int x = 10; x < 20; x = x+1)
   {
      cout<<"value of x : " + x" ;
      cout<<"\n";
    }
return 0;
}
```

This would produce following result:

```
value of x : 10
```

```
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

Comparison between Switch & If-Else

- ✓ The If-Else statement is used for manipulating multiple conditions whereas Switch touches upon only one conditional variable and different value of the variable passes the control to the required case.
- ✓ The if-else statement uses both integer as well as floating type values whereas the Switch statement only accepts integer values.
- ✓ In Switch statement all the branches are executed depending on the value of the same variable whereas if-else follows usage of long instruction based statements and may create confusion within blocks.

JUMP STATEMENTS

1. **The break Keyword:** The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement. The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block. Hence the break statement skips the rest of the loop and jumps over to the statement following the loop.

Syntax: The syntax of a break is a single statement inside any loop:

```
break;
```

```
while (test expression)
{
……..
break;
……..
}
statement x;
```

```
for
(initialize;expression;update)
{
……..
break;
……..
}
statement x;
```

```
do
{
……..
break;
……..
} while (expression);
statement x;
```

Example:

```
#include<iostream>
using namespace std;
int main()
{
```

```
    for(int x=10;x<60;x+=10)
    {
      if( x == 30 )  break;
      cout<< x<<endl;
    }
  return 0;
  }
```

This would produce following result:

```
10
20
```

2. **<u>The continue Keyword</u>:** The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop. The continue statement skips the rest of the loop statements and causes the next iteration of the loop. Hence a continue statement will just abandon the current iterations and let the loop start the next iteration.
   - In for loop, the continue keyword causes flow of control to immediately jump to the update statement.
   - In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

Syntax: The syntax of continue is a single statement inside any loop:

```
continue;
```

```
while (test expression)
{
……..
continue;
……..
}
statement x;
```

```
for
(initialize;expression;update)
{
……..
continue;
……..
}
statement x;
```

```
do
{
……..
continue;
……..
} while (expression);
statement x;
```

Example:

```
#include<iostream>
using namespace std;
int main()
{
    for(int x=10;x<60;x+=10)
   {
      if( x == 30 )  continue;
      cout<<x<<endl;
    }
  return 0;
```

```
}
```

This would produce following result:

```
10
20
40
50
```

3. **exit() Function:** C++ provides a way to leave a program before its natural finish with the exit( ) function.
   Syntax : **exit (status) ;**
   where status is an integer variable or constant.

The header file for exit ( ) is : stdlib.h. By convention, an exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened. The value 0 is returned by operating system. The exit() function has no return value of its own.

Eg:

```
#include <iostream>
#include< stdlib >
using namespace std;
int main( )
{
exit(0); // force program to end here
cout<<"\n C++ programming";
cout<<"\n Computer science";
return 0;
}
```

This program will produce no output because the program faces an early exit using exit(0).

4. **The goto statement:**

Goto allows making an absolute jump to another point in the program. We should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations. The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).

Syntax:

goto label; label:	label: goto label;

Generally speaking, this instruction has no concrete use in structured or object oriented programming but used heavily in low-level programming.

Example:

```
// goto loop example
#include <iostream>
using namespace std;
int main ()
{
int n=10;
loop: cout << n << ", ";
      n--;
      if (n>0) goto loop;
      cout << "FIRE!\n";
      return 0;
}
```

Output: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

UNIT III

Functions - Library functions - User-defined functions - Function declarations and definitions - Passing arguments - Inline functions - Overloading.

FUNCTIONS

A self-contained collection of steps that solves a problem or sub-problem is called as module. Large programs can be made manageable by dividing it into smaller subprograms or modules. These subprograms are called as functions. A function is a named unit of a group of program statements designed to perform a specific task and returns a single value. A function is a group of statements that is executed when it is called from some point of the program.

Reasons for using functions:

- ✓ Programs with functions are compact thus decreasing the number of lines (avoids repetitive code).
- ✓ Debugging the code is much simpler because errors can be localized and corrected easily.
- ✓ Functions increase the readability and documentation of program.

There are two types of functions as Library Functions and User-Defined Functions.

LIBRARY FUNCTIONS

The standard C++ library is a collection of pre-defined functions and other program element which are accessed through header files (definitions of the library functions are held here). They are built-in functions whose definitions are already given. The names and arguments taken by that library function are already written. The programmer has to call the function by just writing its name along with necessary parameters. The two popular set of library functions are mathematical library functions and string library functions, which are defined at cmath and cstring header files respectively.

Name	Description
acos	Returns the arc cosine value.
atan	Returns the arc tangent value.
cos	Returns the cosine value.
ceil	Returns the largest integer not less than the given value.
exp	Returns the natural logarithm base "e" raised to the argument value.
fabs	Returns the absolute value of the num.
floor	Returns the smallest Integer not greater than the given value.
log	Returns the natural algorithm for the given value.
log10	Returns the base 10 algorithm for a number.
pow	Returns the exponential value.
sin()	Return the sine value for the argument.
sqrt()	Returns the square root of the argument.
tan()	Return the tangent value for the argument.

Standard Function Library has a numerous set of **string and character handling functions**. The string functions operate on the null-terminated arrays. The header file required for string function is "**<cstring.h>** and for character use the header file "**<cctype.h>**".

Name	Description
Character Type Functions	
isalnum()	Checks if the given argument is alphanumeric character.

isalpha()	Checks if the given argument is an alphabet.
isdigit()	Checks if the given argument is a digit(0-9)
ispunct()	Checks if the given argument is a punctuation character
iscntrl()	Checks if the given argument is a control character
isspace()	Checks if the given argument is a space character
isupper()	Checks if the given argument is a uppercase alphabets
islower()	Checks if the given argument is a lowercase alphabets
isxdigit()	Checks if the given argument is a hexadecimal digit
tolower()	Returns the ASCII value of lowercase of the parameter.
toupper()	Returns the ASCII of uppercase of the parameter.
The above functions accept only character as input	
String Functions	
strcpy()	Copies the content of a string to another.
strcat()	Appends a copy of the source string to the destination string
strcmp()	Compares two strings
strlen()	Returns the length of a string.

USER DEFINED FUNCTIONS

Using functions we can structure programs in a more modular way. Each function is manipulated in a given program by having three approaches:

1. **Function Prototype Declaration**- C++ treats a function as a normal variable. So it expects a function should also be declared as a variable before its usage. The declaration of the function will be
   Return type function_name ([datatype parameter1, data type parameter2, ..].); It is enough to specify only parameters' datatype and parameters's name is not needed (ie)**Return type function_name ([datatype, datatype, ...]);** will satisfy a function prototype declaration. It enables a compiler to compare each use of function with the prototype to determine whether the function is invoked properly or not. The number and type of arguments can be easily compared and any wrong number or type of the argument is reported.
2. **Function Call** – It is the statement which actually invokes the function. It will be **function_name([parameter1,parameter2,.....parametern]);** In a

function call, parameters' name should only be specified and the data type of the parameter should not be explicitly written here.

3. **Function Definition**- It denotes the expansion of function (ie) writing the statements of functions (body of function).

   Syntax:

   ```
   Return type function_name ([ datatype parameter1, data type parameter2, ...])
   {
       statements;
   }
   ```

   where,

   - ✓ returntype is the data type specifier of the data returned by the function.
   - ✓ function_name is the identifier by which it will be possible to call the function.
   - ✓ Parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable.

Points to Ponder:

1. The word parameters and arguments can be used interchangeably.
2. The arguments present in the function call are called as actual arguments.
3. The arguments present in the function definition or header are called as formal arguments.
4. Number of actual arguments=Number of formal arguments (except default arguments).
5. Data type of the actual arguments = Data type of the corresponding formal arguments.
6. Actual arguments can be constant while formal argument must be a variable.
7. It is optional to specify void in the parameter list (ie) a parameter list can simply be left blank if we want a function with no parameters.

How function works?

1. Define the user-defined function either before or after the main program.
2. There are now two functions – Calling function (usually main) and Called function (user-defined function).
3. Function call calls the function by passing arguments if any.
4. The statements of the called function are now executed. After performing the statements, the called function may or may not return value to the calling function.
5. After the completion of the called function, the calling function will continue its execution.

Local and Global variables:

Variables declared inside the block and not accessible outside of that block are called local variables. The scope of the local variables is confined only within that function. The variable name may be given in another function and each variable will be treated as a different entity.

Variables declared outside the main function are called global variables. The values of the global variables are available throughout the program and in every block of the program, with same data type and same name.

Types of Functions (according to passing arguments and return type):

On the basis of return type and arguments, the functions can be classified as follows:

a. **Function with no arguments and no return value** – Here the calling program will not give any arguments to called function and the latter will not return any value to the former. The return type must be void.

```
function1()                    function2()
{function2();                  {
}                              }
```

Eg:

```
//Addition of two numbers
#include<iostream>
using namespace std;
int main()
{
void add();
add();
return 0;
}
void add()
{
int x,y;
cout<<"Enter 2 integers:";
cin>>x>>y;
cout<<"Sum="<<x+y;
}
```

a. **Function with arguments and no return value –** Here the calling program may pass arguments to the called function but the latter will not return any value to the former. The return type must be void.

```
function1()                    function2()
{                              {
function2();
```

```
}                                   }
```

Eg:

```
//Addition of two numbers
#include<iostream>
using namespace std;
int main()
{
void add(int,int);
int x,y;
cout<<"Enter 2 integers:";
cin>>x>>y;
add(x,y);
return 0;
}
void add(int a,int b)
{cout<<"Sum="<<a+b;}
```

b. Function with no arguments and return value – Here the calling function will not pass any arguments to the called function but the latter returns one value to the former.

```
function1()                         function2()
{                                   {
function2();
}                                   }
```

Eg:

```
//Addition of two numbers
#include<iostream>
using namespace std;
int main()
{
int add();
cout<<"Sum="<<add();
return 0;
}
int add()
{
int x,y;
cout<<"Enter 2 integers:";
cin>>x>>y;
return (x+y);
```

}

c. **Function with arguments and return value** – Here the calling function will pass arguments to the called function and the latter also returns one value to the former.

```
function1()                    function2()
{                              {
function2();
}                              }
```

Eg:

```
//Addition of two numbers
#include<iostream>
using namespace std;
int main()

{
int add(int,int);
int x,y;
cout<<"Enter 2 integers:";
cin>>x>>y;
cout<<"Sum="<<add(x,y);
return 0;
}
int add(int a,int b)
{ return (a+b);}
```

Types of passing arguments:

There are two types of passing arguments or parameters to functions. They are Call By Value and Call By Reference.

1. ***Call By Value*** – This means the values at the actual arguments in the function call are just copied to the formal arguments in the function header. So any change at formal arguments occurs in the function body will not affect the values at actual arguments because two separate memory locations are created separately for actual and formal arguments.

2. ***Call By Reference*** – Here the formal parameters will act as a synonym for actual arguments. So any change at formal arguments occurs in the function body affect the values at actual arguments because a single memory location is used for both actual and formal arguments.

Eg for Call by value:

```
#include<iostream>
using namespace std;
int main()
{
void swap(int,int);
int x,y;
cout<<"Enter 2 integers:";
cin>>x>>y;
swap(x,y);
cout<<"x="<<x<<"y="<<y;
return 0;
}

void swap(int a,int b)
{
int t=a;
a=b;
b=t;
}
```

Eg for Call by reference:

```
#include<iostream>
using namespace std;
int main()
{
void swap(int &,int &);
int x,y;
cout<<"Enter 2 integers:";
cin>>x>>y;
swap(x,y);
cout<<"x="<<x<<"y="<<y;
```

```
return 0;
}
void swap(int &a,int &b)
{
int t=a;
a=b;
b=t;
}
```

RECURSIVE FUNCTION

A function is recursive (or has a recursive definition) if the definition includes a call to itself (function calling itself is called as a recursive function). Recursion is a familiar idea in mathematics and logic. For example, the natural numbers themselves are usually defined recursively. Very roughly speaking, the definition is:

- 0 is a natural number.
- if n is a natural number then s(n) (i.e. n+1) is a natural number, where s is the "successor function".

In this context, the notion of recursion is clearly related to the notion of mathematical induction. A recursive function must have 2 parts:

a) Base case= Case for which the solution can be stated non-recursively.
b) Recursive case= Case for which the solution is expressed in terms of recursion.

Notice also that the above definition includes a non-recursive part or base case (the statement that 0 is a natural number). Another familiar mathematical example of a recursive function is the factorial function "!". Its definition is:

- ✓ 0! = 1
- ✓ for all n > 0, n! = n(n-1)!

Thus, by repeatedly using the definition, we can work out that 6! = 6x5! = 6x5x4! = 6x5x4x3! = 6x5x4x3x2! = 6x5x4x3x2x1! = 6x5x4x3x2x1x1 = 720

Eg:

```
#include<iostream>
using namespace std;
int main()
{
int fact(int); int n;
cout<<"Enter an integer=";
cin>>n;
cout<<"Factorial ="<<fact(n);
return 0;
```

```
}
int fact(int x)
{
if (x==1)
  return (1);
else
 return (x*fact(x-1));
}
```

Enter an integer=4
Factorial=24

INLINE FUNCTION

A function call involves substantial overhead. Extra time and space have to be used to invoke the function, pass parameters, allocate storage for its local variables, store the current variables and the location of execution in the main program etc. Instead of transferring the control to and from the function code segment (prototype, calling statement, definition), inline function copies the function body and substituted directly for the function call. An inline function comparatively takes less time to execute. An inline function is a function that is expanded in a line. By adding the keyword "inline" before the function definition, a function is turned as an inline function. The speed benefits of inline function diminish as the function grows in size. So the functions are made inline when they are small enough to be defined in one or two lines. It should be noted that the inline keyword merely sends a request, not a command to the compiler. The compiler may ignore the request if the condition is not met for a function to be an inline.

Syntax:

```
inline return type function name(parameter list)
{

}
```

When we should make a function inline?

(1) The function is small.
(2) The function does contain a loop, switch or a goto.
(3) It is not containing static variables and not recursive.

Eg:

```
//Inline functions
#include<iostream>
using namespace std;
int main()
```

```
{
int x,y;
cout<<"Enter 2 integers:";
cin>>x>>y;
cout<<"Addition="<<add(x+y);
return 0;
}
inline int add(int n1,int n2)
{
return (n1+n2);
}
```

FUNCTION OVERLOADING

"Overloading" refers to the use of the same thing for different purposes. C++ allows using a same name to use across several functions called as **function overloading or function polymorphism**. To distinguish them, they must have different return type, different number of arguments or different data types of arguments. These three together are referred to as the **function signature**. We should not overload unrelated functions and should reserve function overloading for functions that perform closely related operations. By function overloading, we can design a family of functions with one function name but with different argument lists or return type. Function overloading not only implements polymorphism but also reduces number of comparisons in a program and thereby makes the program run faster. The correct function to be invoked is determined by checking the function signature.

Eg:

```
void f1(int);
void f1(char);
int f1(int,int);
float f1(int);
```

The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are same.
2. If exact match is not found, the compiler uses the integral promotions to the actual arguments as char, short, enumerator to int, float to double to find a match.
3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique ((i.e,) int to long or double).

Eg:

```
//Function overloading
```

```
#include<iostream>
using namespace std;
int main()
{
int max(int,int), max(int,int,int),x,y,z;
cout<<"Enter 3 integers:";
cin>>x>>y>>z;
cout<<"Biggest between x and y="<<max(x,y);
cout<<"Biggest among x,y,z="<<max(x,y,z);
return 0;
}
int max(int a,int b)
{
int big;
big=(a>b)?a:b;
return (big);
}

int max(int a,int b,int c)
{
int big;
big=(a>b)?a:b;
big=(big>c)?big:c;
return (big);
}
```


UNIT IV

Arrays - Defining and Processing - Initializing - Passing Arrays to Functions - Enumeration type - Multidimensional arrays - String Manipulation - Strings and Arrays - Structures and Unions - Passing Structures to Functions - Structure within Structure.

ARRAYS

An array is a collection of variables of the same type that are referenced by a common name. It occupies contiguous memory locations. We can have a direct access or random access over any individual element in an array.

Need for arrays:- For large applications or calculations say to find the average marks of 50 students if we use 50 different variables the job will be cumbersome and tedious and program maintenance will be very difficult. But if we use array the

remembering and managing these variables will be easy. To make the program complex –free and more understandable array is used.

Declaration of arrays:

data type array-name[size];

Eg: int A[10]; float B[20]; char C[30];

where data type refers to any fundamental(or primitive or in-built) data types, array-name will be any valid identifier name and size will be any number which indicates number of elements the array contains.

Types of Arrays: - Arrays are of two types:-

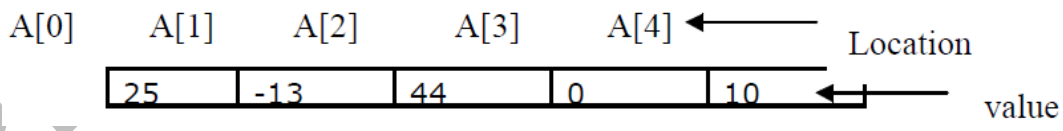
(i) One-dimensional arrays
(ii) Two-dimensional arrays.
Both the arrays can be of numeric as well as character types.

ONE-DIMENSIONAL OR SINGLE DIMENSIONAL OR 1-D ARRAYS

It is a collection of similar types of elements referenced by a common name. The elements of the array are referred by their subscript or index. Subscript represents the size of the array and it should always be an integer constant.
Eg: int A[5];

The above array will be having elements : A[0], A[1], A[2], A[4], A[5]. Array subscripts (the numbers which are within [ ]) always starts with number 0 not 1 and spans up to size-1 in C++. It should be noted that C++ will never perform boundary checking and so it is the task of the programmer to manipulate within 0 to size-1 for the given array. Subscripts are the locations of the elements within the array.

Memory representation of 1-D array: The memory representation for int A[5];



where A[0] is the first element, A[1] is the second element and so on. The last element (i.e. 10 here) always having location A[size-1]. Each element contains 2 bytes of memory. The address of first element of the array i.e. &A[0] is called the base address of the array.

Note: - The name of the array (here A) is a pointer to its base address i.e. first element's address (i.e. &A[0]).

Size of the array in memory = data-type X size of the array

For the above array the memory occupied will be 2 X 5 = 10 bytes. We can find it using C++ coding also cout << sizeof(A); here sizeof is a operator which is used to find the size of any data type, variable or derived data type in bytes and A is the array name.

Initialization (numeric type): There are two types of initialization

(i) Sized array initialization and

(ii) Unsized array initialization

(i) Sized array initialization: - It can be done during the program writing (i.e. before program execution) or during the program execution (Using Loop). This is called as direct initialization.

Eg: int A[5] = {12, 13, 20, 5, 2};

float B[4] = { 3.4, 4.5, 6.2, 7.8};

Code Segment:

```
int A[5];
for( int i=0; i<5; i++)
        cin >> A[i];
```

Here the values must be provided by the user during the execution of the program. In the first case the values are actually assigned during the compilation time and in the second case the values are assigned during the execution time by the user's input.

(i) Unsized array initialization: - It can only be done in following ways:

int A[ ] = {2, 5, 17, 10, 12}; // No size is given within [ ]

char str[ ] = "C++ Programming";

The compiler automatically fixes the size of the array. Here the advantage is that the number of elements can be increased or decreased without considering the size. It is more flexible whenever there is a requirement of direct initialization or the values are known to the user.

String as an array:- C++ does not string data types rather it implements string as one-dimensional character arrays. String is defined as an array of characters that is terminated by a null character '\0'. A group of characters defined within double quotes is also considered as string in C++. For storing the null character, the size of the string must be equal to the maximum number of characters plus one else the last character will be lost and it will be replaced by null character.

Eg: char str[10] = "India";

It can be represented as

I	n	d	i	a	\0				

The above array contains 5 characters, but it can contain maximum of 10 characters. It occupies 6 characters i.e. one space is reserved for null character. But the length of the array is 5. The position of null character is not taken for finding the length of the array.

TWO-DIMENSIONAL (2-D) ARRAYS

A two-dimensional array is a multi one-dimensional array. It consists of rows and columns i.e. it is in a matrix format. For instance an array A[m][n] is an m by n table with m rows and n columns containing m X n elements.

The number of elements = number of rows X number of columns

Declaration of 2-D array:

The general form of a two-dimensional array is

data type array-name [rows] [columns];

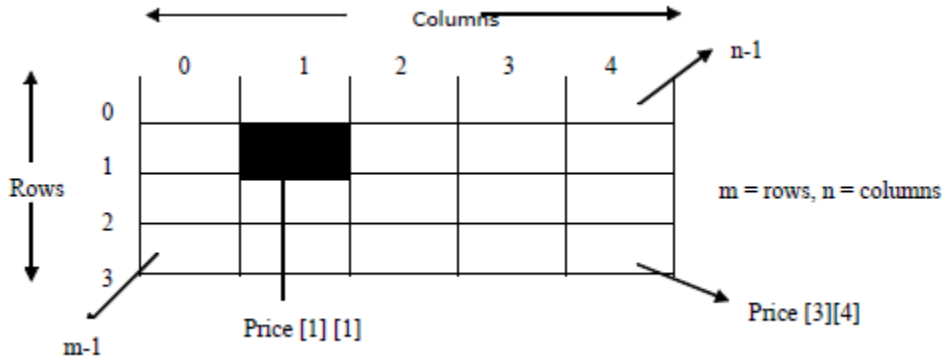
where data-type is any base data type, array-name is any valid identifier name and rows, the first index indicates the number of rows and columns the second index, indicates the number of columns.

Eg: int Price [4] [5];

The array Price have 4 elements Price[0], Price[1],……., Price[3] which itself an int array with 5 elements. The individual elements of Price are referred to as Price[0][0], Price[0][1],……., Price[0][3], Price[1][0], Price[1][1] and so forth. The last element will be Price [3][4].

Memory Map Representation:-

Ex:- int Price [4][5];



Size occupied by the above array in memory = data-type X rows X columns

= 2 X 4 X 5 = 40 bytes

C++ statement will be : cout << sizeof(Price);

Array of Strings or Table of Strings:- The two-dimensional character array is known as array of strings. The size of the first index (rows) is the number of strings and the size of the second index (columns) is the maximum length of each string.
Eg: char days [7][11];

declares an array of 7 strings each of which can hold maximum of 10 valid characters where 1 extra to take care of the null character '\0'. The above array of strings appears in memory as shown below:

0 M O N D A Y \0
1 T U E S D A Y \0
2 W E D N E S D A Y \0
3 T H U R S D A Y \0
4 F R I D A Y \0
5 S A T U R D A Y \0
6 S U N D A Y \0

The above array of strings appears in memory as shown below:

(11)

(7)	0	1	2	3	4	5	6	7	8	9	10
0	M	O	N	D	A	Y	\0				
1	T	U	E	S	D	A	Y	\0			
2	W	E	D	N	E	S	D	A	Y	\0	
3	T	H	U	R	S	D	A	Y	\0		
4	F	R	I	D	A	Y	\0				
5	S	A	T	U	R	D	A	Y	\0		
6	S	U	N	D	A	Y	\0				

Array Initialization:- Like One-dimensional array, Two-dimensional array can be initialized in two ways:-

(i) Sized 2-D array initialization

```
int SQ[3][2] = { 1, 1,
                 2 , 4,
                 3 , 9
                };
int SQ[3][2] = {1,1, 2,4, 3,9};
```

For Array of strings: char months[12][4] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul","Aug", "Sep", "Oct", "Nov", "Dec"};

(i) Unsized 2-D array initialization

```
int Cube[ ][2] = { 1, 1,                    //rows should be left blank
                   2 , 8,
```

```
                    3 ,27,
                    4 , 64
                  };
```

char months[ ][4] = {“Jan”, “Feb”, “Mar”, “Apr”, “May”, “Jun”, “Jul”, “Aug”, “Sep”, “Oct”, “Nov”, “Dec”};

The advantage is that one can increase the rows as and when required. In C++ program nested loop is required for taking inputs for numeric type 2-D arrays whereas generally single loop is used for taking inputs in character type 2-D arrays (i.e. for array of strings). For string input gets ( ) function (which is defined in stdio.h header file) is normally preferred over cin>> operator since it can able to take the white space characters (tabs, spaces etc).

Code segment: int mat[4][5];

```
for (int i = 0; i < 4; i++)
        for (int j = 0; j < 5; j++)
cin >> mat[i][j]; - - Here i indicates rows & j indicates columns
char days [7][11];
for (int i = 0; i < 7; i++)
    gets(days[i]);
```

PASSING ARRAYS TO FUNCTIONS

When an array argument is passed to a function, C++ handles it differently. C++ treats the array name as a pointer i.e., only the address of the array gets passed, not a copy of the entire array. The pointer of the first element in the array is passed into the function. So any change made in the array elements within the function is also reflected in the main program also. There are three ways to declare a parameter that is to receive an array pointer.

First way: Receiving parameter may also be declared as a sized array itself.

Eg:

```
#include<iostream>
using namespace std;
int main()
{
int a[10];
void display(int x[10]);
cout<<”Enter 10 elements:”;
for(int i=0;i<10;i++)
  cin>>a[i];
display(a);
```

```
return 0;
}
void display(int x[10])
{
for(int i=0;i<10;i++)
   cout<<x[i]<<endl;
}
```

Here the parameter x is declared as an int array of 10 elements, the C++ compiler automatically converts it to an int pointer. This is very much necessary because no parameter can actually receive an entire array. When an array is passed, actually a pointer to an array gets passed and a pointer parameter can only receive it.

Second way: The receiving parameters array may be declared as an unsized array. Since C++ converts an array declaration into an array pointer, the actual size of the array is irrelevant to the parameter. **NOTE**: While passing a multidimensional array as an argument, its size must be specified. However, the size of the first dimension is optional.

Eg:

```
#include<iostream>
using namespace std;
int main()
{
int a[10];
void display(int x[]);
cout<<"Enter 10 elements:";
for(int i=0;i<10;i++)
   cin>>a[i];
display(a);
return 0;
}

void display(int x[])
{
for(int i=0;i<10;i++)
   cout<<x[i]<<endl;
}
```

Third way: The receiving parameter can be declared as a pointer. This is allowed since a pointer receiving an array can be used as an array. Even though arrays and pointer are closely linked, they are different things. The only exception is for function arguments passed as arrays.

Eg:

```
#include<iostream>
using namespace std;
int main()
{
int a[10];
void display(int *);
cout<<"Enter 10 elements:";
for(int i=0;i<10;i++)
   cin>>a[i];
display(a);
return 0;
}

void display(int * x)
{
for(int i=0;i<10;i++)
   cout<<x[i]<<endl;
}
```

STRUCTURES

So far we used arrays to represent a group of items that belong to the same type. If we are in a situation to group data items of various data types, array will not be helpful. There is a constructed data type called **structure** which packs variables belonging to various data types. It is convenient to handle a group of logically related items via structure. Hence structure is a collection of heterogeneous variables of different data types referenced by a name. A structure is analogous to record.

Syntax:

Structure definition

```
struct tagname
{
datatype member1;
datatype member2;
……..
datatype membern;
};
```

Structure declaration

```
struct tagname var1,var2,……varn;
```

NOTE:

1. The template is terminated by semicolon. Definition starts with the keyword 'struct'.
2. Even though the entire declaration can be taken as a statement, each member is declared independently with its name and type inside the template. Structure members cannot act independently.
3. It is also allowed to combine both the template and variable declaration in one statement as,
   struct book_bank
   {
   int num;char name[20];int pages; float price;
   } book;
4. Tag name is optional (i.e.)
   struct
   {
   int num;char name[20];int pages; float price;
   } book; is also valid.
5. Definition does not create memory space for structure variable instead it is done only after declaration.

STRUCTURE INITIALIZATION

The member of a structure can be accessed by dot operator '.' as

Structure variable.member.

No initialization should be done for individual member inside the template. Instead they can be initialized as follows:

1. struct sample
   {int x;float y;char z};
   struct sample s={10,5.6,'a'); is valid.
2. struct sample
   { int x;
   float y;
   char z} s={10,5.6,'a'); is valid.
3. struct sample
   { int x=10;
   float y=5.6;
   char z='a'}; is not valid.
4. We can also read the values for the structure members using cin as s.x, s.y and s.z.

STRUCTURES TO FUNCTIONS

C++ supports the passing of structure values as arguments to functions. There are three methods by which the values of a structure can be passed to a function as,

1. Pass each member of the structure as an actual argument to the function call. They are treated as individual variables. The function can either receive the values by creating its own copy for them (call by value) or by creating references for the original variables (call by reference). But is an elementary methods and also inefficient and unmanageable when the structure size is large. **Eg:** func(s.num,s.name,s.qty);
2. Pass the copy of the entire structure to the called function. Here also the entire structure can be passed to the functions both ways by value and by reference. Passing by value is useful when the original values are not to be changed. Passing the structure variable by reference variable (call by reference). When a structure is passed by reference the called function declares a reference for the passed structure and refers to the original structure elements through its reference.

Eg:

```
//Call by Value
#include<iostream>
using namespace std;
int main()
{
        struct stores
        {
        int num,qty;
        float up,amt;
        }s;
        struct stores assign();
        s=assign();
        cout<<"Product Number="<<s.num;
        cout<<"Quantity="<<s.qty;
        cout<<"Unit Price="<<s.up;
        cout<<"Amount="<<s.amt;
        return 0;
}
struct stores assign()
{
struct stores p;
cout<<"Enter product number, quantity, unit price:";
cin>>p.num>>p.qty>>p.up;
p.amt=p.qty*p.up;
return (p);
```

}

3. The third way to pass arguments to functions is to use pointers.

STRUCTURE WITHIN STRUCTURE or NESTED STRUCTURE

The members of a structure may be either complex or simple. The simple elements are any of the fundamental data types of C++ as int, char float or double. But there is also a possibility of having complex data type as arrays, structures etc. Hence structure within structure is called as nesting of structures. Here one structure is defined as a member within another structure.

Eg:

```
struct salary
{
        int num;
        char name[30];
        char dept[30];
        float basic;
        struct pay
                {float da,hra,cca;} allowance;
} emp;
```

The salary structure contains a member called allowance which itself is a structure with 3 members da,hra and cca. The members of emp can be accessed as:emp.num, emp.name,emp.dept, emp.allowance.da, emp.allowance.hra and emp.allowance.cca. The above nested structure can be rewritten as:

```
struct pay
        {float da,hra,cca;};
struct salary
{
int num;
char name[30];
char dept[30];
float basic;
struct pay allowance;
} emp;
```

Eg:

//Structures within structure

```
#include<iostream>
using namespace std;
int main()
{
struct salary
        {
                int num;
                char name[30];
                char dept[30];
                float basic;
                struct pay
                        {float da,hra,cca;} allowance;
                float pf,gross,net;
        } emp;
cout<<"Enter the employee number:";
cin>>emp.num;
cout<<"Enter the employee name:";
cin>>emp.name;
cout<<"Enter the employee department:";
cin>>emp.dept;
cout<<"Enter the employee basic pay:";
cin>>emp.basic;
emp.allowance.da=basic*1.07;
emp.allowance.hra=basic*0.08;
emp.allowance.cca=basic*0.02;
emp.pf=basic*0.12;
emp.gross= emp.basic+emp.allowance.da+emp.allowance.hra+emp.allowance.cca;
cout<<"\n Gross pay of the employee=";
cout<<emp.gross;
emp.net=emp.gross-emp.pf;
cout<<"\n Net pay of the employee="<<emp.net;
return 0;
}
```


UNIONS

Unions are the concept borrowed from structures and therefore they follow the same syntax as structures. There is a major difference between them in terms of storage. In structures, each member has its own storage location. In union, all the members are going to share the same location (ie) a union may contain many members of different type, it can handle only one member at a time. To declare a union, prefix the keyword 'union' as,

```
union item
{
        int m;
        float x;
        char c;
} code;
```

The above union contains 3 members with 3 different data types. The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the above example, 'x' requires 4 bytes, largest of all.

1st byte	2nd byte	3rd byte	4th byte

To access the members of union, use the same syntax as per structure as, **code.x, code.m and code.c**. During accessing, we should make sure that we are accessing the member whose value is currently stored.

Eg:

```
code.m=101;
code.x=56.67;
cout<<code.m; will produce error. It may be rewritten as,

code.m=101;
temp=code.m;
code.x=56.67;
cout<<temp;
```

Nesting of union within structure is allowed. Therefore union is a mutually-exclusive collection of elements of which only one is used at any given instant of time.

Eg:

```
//Program for union
#include <iostream>
using namespace std;
 int main()
{
union emp
{
int num;
double sal;
```

```
} value;
value.num = 2;
cout << "Employee Number::" << value.num<<endl;
cout<< "\nSalary is:: " << value.sal << endl;
value.sal = 2000.0;
cout << "Employee Number::" << value.num<<endl;
cout<< "\nSalary is:: " << value.sal << endl;
return 0;
}
```

Output:

```
Employee number is::2
Salary is::2.122e-314
Employee number is::0
Salary is::2000
```

ENUMERATION TYPE

C++ allows us to define own user-defined data types. An enumeration type is an integral type that is defined by user as follows:

enum typename {enumerator list}; where

1. enum=keyword
2. typename=identifier
3. enumerator list=list of names for integer constants

The values in the enumerator list are called as **enumerators** which are ordinary integer constant whose value will be automatically assigned as 0,1,2,3…n.

Eg:

```
enum season {WINTER, SUMMER, MONSOON, SPRING};
```

NOTE:

i) Enumerators should be a valid C++ identifier.

Eg: enum RESULT {D, A+, A, B+, C}; is not correct because A+ and B+ are not valid C++ identifiers.

ii) Anonymous enumerators are allowed in C++. They are useful in declaring integer constants.

Eg: enum {I=1,V=5,X=10,L=50,C=100,D=500,M=1000};

iii) enum DAY { SUN=1, MON,TUE,WED,THU,FRI,SAT}; It will assign the values from 1 to 7 i.e., if any integer value is assigned to some enumerator, the others that follow are given with consecutive values.

iv) enum ANSWER {NO=0,FALSE=0,YES=1,TRUE=1}; It is legal to define different enumerators with the same value.

v) Enumeration should not be overused (i.e.,) the enumerator list defines a new set of identifiers. Care should be taken that these identifiers should not be used anywhere in the same program.

Eg:

```
enum BASE{BIN=2,OCT=8,HEX=16,DEC=10};
enum MONTH{JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC};
```

vi) Enumeration is usually defined to make the code more self-documenting and readable.

UNIT-5

Pointers - The Reference Operator - References - The Dereference Operator - Derived types - Arrays and Pointers - Dynamic arrays - Arrays of Pointers - Pointer to arrays - Pointer to Pointers - Pointer to Functions.

POINTERS

Computer memory can be imagined as a large array of bytes. Computer use their memory for storing the program and also the values of variables. Each array of bytes is called as cell has a number called address which is numbered consecutively starting with 0. A computer system having 64k memory will have its last address as 65,535. A variable declaration associates three fundamentals attributes to the variable:-

- Its name
- Its data type
- Its memory address

Eg: int n= 179; n➔ variable name
179➔value
5000➔address

During execution of the program, the system always associates the name 'n' with the address 5000, to having a house number as well as house name. We have the access to value 179 either by n or by 5000. Since memory address is simply numbers, they can be assigned to some variables which can be stored in memory. Such variable which holds the address of other variables is called pointer. Three reasons to use pointer are:

1. Provide a means through which the memory location of a variable can be directly accessed.
2. Supports dynamic allocation routines by two operators-new and delete.
3. Improves the efficiency of certain routines.

Since a pointer is a variable, its value is also stored in the memory in another location. Let we assign the address of n to a variable p.

Variable value address

n	179	5000
p	5000	5048

Pointers are one of the strongest and also one of the most dangerous and vulnerable features of C++. Uninitialized or wild pointers can cause the system to crash.

THE REFERENCE OPERATOR

We can obtain the address of the variable by using the Reference operator & (same as bitwise AND) which is also called as address operator.

Eg:-
```
//Pointers
#include <iostream>
using namespace std;
int main ()
{
 int n=44;
 cout<<"n="<<n<<"address of n="<<&n<<endl;
 return 0;
}
```

THE DEREFERENCE OPERATOR

Every variable must be declared for its type. Since pointer variables contain address that belongs to a separate data type, they must be declared before its use. The declaration of a pointer variable will be,

data type *pointer name;

Note:-

- '*' tells the variable is a pointer.
- Pointer name has a data type i.e it can point to a variable belonging to the same data type.
- '&' operator denotes the address of the variable.

Eg:- int * x; makes a pointer 'x' to point only another integer variable. Once a variable has been declared we can make it to point another variable by symbol.ie.

```
int n,*x;
x=&n;
```

This is called a **Pointer Initialization.** Always ensure that pointer variables always point to the corresponding type of data.

Eg:- float a;
int *p;
p=&a; will produce wrong results.

Assigning an absolute address to a pointer variable is prohibited.
i.e p=5068; is wrong.
int *x=&n,n; will also be wrong.

Eg :- //Pointers

```
# include <iostream>
using namesapace std;
int main()
{
int n=44;
cout<<"n="<<n;
cout <<"Address of n="<<&n;
int *pn=&n;
cout<<"pn="<<pn;
cout<<"&pn="<<&pn;
cout<<"*pn +"<<*pn;
return 0;
}
```

Output: n=44
address of n = 0*0064fdd3
&pn=00064fde0
*pn=44

Here *pn will evaluate the value of n and it is called as **deferencing the pointer** and symbol *is called the **dereference operator**. The reference operator & and the deference operator * are inverses and both have higher precedence than all other arithmetic operators.

REFERENCE VARIABLE

A reference is an alias or synonym for another variable.

Syntax:- type &reference name= variable;
type➔data type
variable➔ should be already declared before reference is created.

Eg:- int &rn =n;

i. Like constants, reference must be initialized
ii. When they are declared, they must be initialized to a variable and not to a literal, int &m=30; is wrong.

iii. Though they are initialized to a variable, references are not actual variables.
iv. References are used mostly for reference parameters.
v. What are the changes made on the variable are reflected on the reference variable and vice versa.

Eg:- //Reference variable

```
#include <iostream>
using namespace std;
int main ()
{
int n=44;
int&m=n;
cout<<"&n="<<&n <<:&rn ="<<&rn <<endl;
--n;
cout<<"n="<<n<<"rn ="<<rn ="<<rn <<endl;
rn *=2;
cout<<"n=" <<n<<"rn= "<<rn;
return 0;
}
```

Output: &n=0*0064fde4 &rn =0*0064fde4
n = 43 rn=43
n =86 rn=86

ARRAY AND POINTERS or POINTER TO ARRAYS

Like other variables, pointers variables can be used in expression. If p1 is a pointer variable, p1++ is valid. The effect of this arithmetic is to cause the pointer to point to another memory location. The actual change in address depends upon the size of the fundamental type to which the pointer points. Pointers can be incremented or decremented like integers- however the increase or decrease in the pointers value is equal to the size of the object to which it points. Generally the lengths of various data types are as follows:-

Characters -1 bytes
Integers – 2 bytes
Float -4 bytes
Double-8 bytes (usually the bytes occupied are system- dependent)

We use this concept of incrementing pointers to relate pointers and arrays. Generally pointers and arrays are closely related. C++ interprets an array name as

the address of its first element. When an array is declared, the compiler allocates a base address and sufficient amount of memory to store all array elements in contiguous memory location. The base address is the location of the arrays first element.

Let int x [5] = {1,2,3,4,5};

Elements ➔x[0] x[1] x[2] x[3] x[4]

Values➔

1	2	3	4	5

Address➔ 1001 1003 1005 1007 1009

Declare p as an integer pointer and make it to point the above array x, as p=&x[0] (or) p=x is enough. Now we can access the elements of x by applying p++. The array subscript operator [] equivalent to the dereference operator *. They provide access into the array the same way.

a[0] =*a

a[1] =*(a+1)

a[2] =*(a+2) etc.

so an array can be traversed like,

```
for (i=0; i<n;i++)
        cout<<*(a+i);
```

ARRAY OF POINTERS

Since pointer is also a variable, pointers may be arrayed like any other data type. An array of 10 integer pointers, the declaration may be,

int *p[5];

After this declaration, 10 contiguous memory locations will be created in memory as,

0	1	2	3	4

Now each of the pointers, the elements of pointer array may be initialized. The name of an array is actually a pointer to the first element of the array, the same holds true for the array of pointers also. All rules that apply to an ordinary array are also applicable for the array of pointers as well.

Eg: - double* p(2);

Its element can be allotted like any other pointer.

p[0] =new double (2.71828);

p [1] = new double (3.14159);

Eg:

```
#include<iostream>
using namespace std;
{
```

```
int *p[5];
int a=11,b=22,c=33,d=44,e=55;
p[0]=&a; p[1]=&b; p[2]=&c;p[3]=&d;p[4]=&e;
for(int i=0;i<5;i++)
   cout<<"\n The value is ="<<*p[i];
for(int i=0;i<5;i++)
   cout<<"\n The address is="<<p[i];
return 0;
}
```

Output:

```
The value is =11
The value is =22
The value is =33
The value is =44
The value is =55

The address is=0x22f72450
The address is=0x22f72452
The address is=0x22f72454
The address is=0x22f72456
The address is=0x22f72458
```

DYNAMIC ARRAY

An array name is really meant the allocation of memory. float x[10]; will allocate 10 blocks of memory to x at Compile time . It is called as **Static Binding**. The allocated memory is bound to x even if the array is never used while the program is running. But if we postpone the allocation of memory until the program is running it is called as **Run-Time Binding or Dynamic Binding** and the array created is called as **Dynamic Arrays**, (i.e,) float *p=new float[10]; The dynamic array p is created only when its declaration executes. Its allocated memory can be de-allocated as soon as the delete operator is invoked. delete[] p; The subscript operator [] must be included because p is an array.

Eg:

```
#include <iostream>
using namespace std;
int main()
{
int i,n; int *p;
cout<<"how many numbers?";
cin>>n;
p= new int [n];
```

```
cout<<"enter numbers ";
for (i=0; i<n; i++)
        cin>>p[i];
cout<<"The numbers are\n";
for (i=0; i<n; i++)
        cout<<p[i] <<"   ";
delete [] p;
return 0;
}
```

POINTER TO POINTER

A pointer may point to another pointer.

Eg :-
```
char c='t';
char*pc=&c;
char**ppc=&pc;
char ***pppc= &ppc;
```

***pppc='w'; this assignment refers to the contents of the address pc that is pointed to by the address ppc that is pointed to by the address pppc. A pointer to a pointer is a form of multiple indirection or chain of pointers.

POINTER TO FUNCTIONS

A function like a variable has an address location in the memory. So it is possible to declare a pointer to a function.

Syntax:-

type (*fptr) (arguments);

Here fptr is a pointer to a function that return type value. The parenthesis around fptr is necessary.

Eg:-
```
{
int f (int);
int (*pf) (int);
pf= &f; (or) pf=f;
cout<<"answer="<<(*pf)(10);
return 0;
}
int f(int x)
{ return (x*10);}
```

POINTER AND STRINGS

We know that the string is an array is an array of characters terminated by null characters. Like 1-D array, we can use a pointer to access the individual characters in a string.

Eg:- //Pointers and Strings

```
#include <iostream>
using namespace std;
int main ()
{
   char *name,*cptr=name;
   int len;
   name =”delhi”;
   for (;*cptr !=null; cptr ++)
            cout <<*cptr <<”is stored at “<<cptr;
    l=cptr-name;
    cout<<”length=”<<l;
   return 0;
 }
```

One important use of pointers is handling of strings.

Eg: - char name [3] [25];

Here name is a table with 3 strings each of 25 characters and so name totally requires 75 bytes we rarely use all this 75 bytes-so in order to make the storage efficient, we can make it a pointer to a string of varying lengths.

Eg:- char* name [3]= {“India”,”Russia”,London”}

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
I	N	D	I	A	NULL	R	U	S	S	I	A	NULL	L	O	N	D	O	N	NULL

This allocation needs only 20 bytes. The character arrays with the rows of varying length are called **Ragged Arrays** and are better handled by pointers.

Note: *p [3] and (*p)[3] are different. Since ‘*’ has a lower precedence than [],*p[3] declares p as an array of 3 pointers while (*p)[3] declares p as a pointer to an array of 3 elements. The array of char pointers is generally preferred over two dimensional arrays of characters because,

1. An array of pointers makes more efficient use of available memory by consuming lesser number of bytes to store the strings.
2. An array of pointers makes the manipulation of the strings much easier. One can easily exchange the positions of strings in the array using pointers without actually touching their memory locations.

DERIVED TYPES

There are five kinds of derived types in C++.

```
const int c=50; //cons int
int & rn=n; //reference variable
```

```
 int *pn =&n;  //pointer pn to n
 int a [] ={33,66};  //array a
int f() //function returning int
{ return (50); }
```

When we define pointer to pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value. A variable that is a pointer to a pointer must be declared as such. This is done by placing additional asterisk in front of its name.

Eg:-

```
#include <iostream>
using namespace std;
int main ()
{
int var;
int *ptr;
int **pptr;
 var=50;
ptr =&var;
pptr =&ptr;
cout<<"value of var ="<<var;
cout<<"value at *ptr ="<<*ptr;
cout<<"value at **pptr  ="<<**pptr;
return 0;
}
```