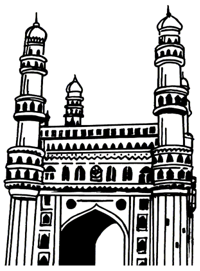
Rahul's ✔
Topper's Voice

AS PER CBCS SYLLABUS



B.Sc.

I Year II Sem

Latest 2020 Edition

PROGRAMMING IN C++

- ☞ Study Manual
- ☞ Lab Programs
- ☞ Short Question and Answers
- ☞ Multiple Choice Questions
- ☞ Fill in the blanks
- ☞ Solved Previous Question Papers
- ☞ Solved Model Papers

` .169/-

- by -

WELL EXPERIENCED LECTURER

Rahul Publications®
Hyderabad. Ph : 66550071, 9391018098

B.Sc.

I Year II Sem

PROGRAMMING IN C++

Price ` 169-00

Sole Distributors : **☎ : 66550071, Cell : 9391018098**

VASU BOOK CENTRE

Shop No. 3, Beside Gokul Chat, Koti, Hyderabad.

Maternity Hospital Opp. Lane, Narayan Naik Complex, Koti, Hyderabad.
Near Andhra Bank, Subway, Sultan Bazar, Koti, Hyderabad -195.

CONTENTS

PROGRAMMING IN C++

SYLLABUS

UNIT - I

Introduction to C++: Applications, Example Programs, Tokens, Data Types, Operators, Expressions, Control Structures, Arrays, Strings, Pointers, Searching and Sorting Arrays.

Functions: Introduction, Prototype, Passing Data by Value, Reference Variables, Using Reference Variables as Parameters, Inline Functions, Default Arguments, Overloading Functions, Passing Arrays to Functions.

UNIT - II

Object Oriented Programming: Procedural Programming verses Object-Oriented Programming, Terminology, Benefits, OOP Languages, and OOP Applications.

Classes: Introduction, Defining an Instance of a Class, Why Have Private Members? Separating Class Specification from Implementation, Inline Member Functions, Constructors, Passing Arguments to Constructors, Destructors, Overloading Constructors, Private Member Functions, Arrays of Objects, Instance and Static Members, Friends of Classes, Member-wise Assignment, Copy Constructors, Operator Overloading.

UNIT - III

Inheritance: Introduction, Protected Members and Class Access, Base Class Access Specification, Constructors and Destructors in Base and Derived Classes, Redefining Base Class Functions, Polymorphism and Virtual Member Functions, Abstract Base Classes and Pure Virtual Functions, Multiple Inheritance. C++ Streams: Stream Classes, Unformatted I/O Operations, Formatted I/O Operations.

UNIT - IV

Exceptions: Introduction, Throwing an Exception, Handling an Exception, Object-Oriented Exception Handling with Classes, Multiple Exceptions, Extracting Data from the Exception Class, Re-throwing an Exception.

Templates: Function Templates–Introduction, Function Templates with Multiple Type, Overloading with Function Templates, Class Templates – Introduction, Defining Objects of the Class Template, Class Templates and Inheritance.

Contents

Topic No. **Page No.**

Topic No. **Page No.**

UNIT I

Introduction to C++: Applications, Example Programs, Tokens, Data Types, Operators, Expressions, Control Structures, Arrays, Strings, Pointers, Searching and Sorting Arrays.

Functions: Introduction, Prototype, Passing Data by Value, Reference Variables, Using Reference Variables as Parameters, Inline Functions, Default Arguments, Overloading Functions, Passing Arrays to Functions.

1.1 Introduction to C++

Q1. What is C++ ?

(OR)

What do you understand by C++?

Ans :

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct feature of Simula67. Since the class was a major addition to the original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented (incremented) version of C.

During the early 1990's the language underwent a number of improvements and changes. In November 1997, the ANSI/ISO standards committee standardised these changes and added several new features to the language specifications.

C++ is a superset of C. Most of what we already know about C applies to C++ also. Therefore, almost all C programs are also C++ programs. However, there are a few minor differences that will prevent a C program to run under C++ compiler. We shall see these differences later as and when they are encountered.

The most important facilities that C++ adds on to C are classes, inheritance, function overloading, and operator overloading. These features enable creation of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

The object-oriented features in C++ allow programmers to build large programs with clarity, extensibility and ease of maintenance, incorporating the spirit and efficiency of C. The addition of new features has transformed C from a language that currently facilitates top-down, structured design, to one that provides bottom-up, object-oriented design.

1.1.1 Applications of C++

Q2. What are the applications of C++ ?

Ans :

1. **Games**

   C++ overrides the complexities of 3D games, optimizes resource management and facilitates multiplayer with networking. The language is extremely fast, allows procedural programming for CPU intensive functions and provides greater control over hardware, because of which it has been widely used in development of gaming engines. For instance, the science fiction game Doom 3 is cited as an example of a game that used C++ well and the Unreal Engine, a suite of game development tools, is written in C++.

2. Graphic User Interface (GUI) based applications

Many highly used applications, such as Image Ready, Adobe Premier, Photoshop and Illustrator, are scripted in C++.

3. Web Browsers

With the introduction of specialized languages such as PHP and Java, the adoption of C++ is limited for scripting of websites and web applications. However, where speed and reliability are required, C++ is still preferred. For instance, a part of Google's back-end is coded in C++, and the rendering engine of a few open source projects, such as web browser Mozilla Firefox and email client Mozilla Thunderbird, are also scripted in the programming language.

4. Advance Computations and Graphics

C++ provides the means for building applications requiring real-time physical simulations, high-performance image processing, and mobile sensor applications. Maya 3D software, used for integrated 3D modeling, visual effects and animation, is coded in C++.

5. Database Software

C++ and C have been used for scripting MySQL, one of the most popular database management software. The software forms the backbone of a variety of database-based enterprises, such as Google, Wikipedia, Yahoo and YouTube etc.

6. Operating Systems

C++ forms an integral part of many of the prevalent operating systems including Apple's OS X and various versions of Microsoft Windows, and the erstwhile Symbian mobile OS.

7. Enterprise Software

C++ finds a purpose in banking and trading enterprise applications, such as those deployed by Bloomberg and Reuters. It is also used in development of advanced software, such as flight simulators and radar processing.

8. Medical and Engineering Applications

Many advanced medical equipments, such as MRI machines, use C++ language for scripting their software. It is also part of engineering applications, such as high-end CAD/CAM systems.

9. Compilers

A host of compilers including Apple C++, Bloodshed Dev-C++, Clang C++ and MINGW make use of C++ language. C and its successor C++ are leveraged for diverse software and platform development requirements, from operating systems to graphic designing applications. Further, these languages have assisted in the development of new languages for special purposes like C#, Java, PHP, Verilog etc.

Q3. Explain about the structure of C++.

OR

Briefly describe about various sections involved in C++ program.

Ans :

Structure of C++ Program

Basically C++ program involves the following sections :

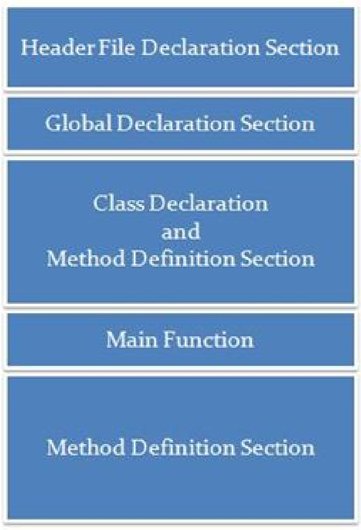

Header File Declaration Section
Global Declaration Section
Class Declaration and Method Definition Section
Main Function
Method Definition Section

Fig. : Structure of C++ Program

Section 1 : Header File Declaration Section

Header files used in the program are listed here.

1. Header File provides Prototype declaration for different library functions.
2. We can also include user define header file.
3. Basically all preprocessor directives are written in this section.

Section 2 : Global Declaration Section

1. Global Variables are declared here.
2. Global Declaration may include :
   - Declaring Structure
   - Declaring Class
   - Declaring Variable

Section 3 : Class Declaration Section

1. Actually this section can be considered as sub section for the global declaration section.
2. Class declaration and all methods of that class are defined here.

Section 4 : Main Function

1. Each and every C++ program always starts with main function.
2. This is entry point for all the function. Each and every method is called indirectly through main.
3. We can create class objects in the main.
4. Operating system call this function automatically.

Section 5 : Method Definition Section

This is optional section . Generally this method was used in C Programming.

1.1.2 Example Programs

Q4. Write the simple Program of C++.

Ans :

```
# include < iostream > // include header file
using namespace std;
int main ( )
{
cout < < "C + + is better than c. \n"; / / C + +
statement
return 0;
}     // End of example
```

1.1.3 Tokens

Q5. Explain briefly about various tokens in C++.

Ans :

C++ tokens are the building blocks in c++ languge. A token is the smallest individual unit of the c++ program. This means that a program is constructed using a combination of these tokens.

They are five main types of tokens in c++

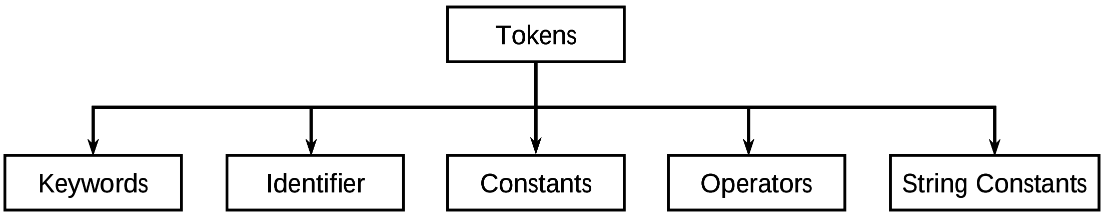


Fig.: C++ Tokens

C++ Character Set

A character set denotes any Alphabet or the Special Symbol that is used for representing the Information Like In C we uses Alphabets, digits and many Types of special Symbols

The following are the character set used in C++.

Letters	A-Z, a-z
Digits	0-9
Special Characters	Space + - * / ^ \ () [] {} = != <> ' " $, ; : % ! & ? _ # <= >= @
Formatting characters	backspace, horizontal tab, vertical tab, form feed, and carriage return

i) Keywords

These are the words used for special purposes or predefined tasks. These words should be written in small letters or lowercase letters.

List of the keyword used in c++ are :

Asm	Else	operator	Template
Auto	Enum	private	This
Break	extern	protected	Throw
Case	Float	Public	Try
Catch	For	register	Typedef
Char	friend	return	Union
Class	Goto	short	Unsigned
Const	If	signed	Virtual
Continue	inline	sizeof	Void
Default	Int	static	Volatile
Delete	long	struct	While
Double	new	switch	-

Keywords cannot be used for the,

1. Declaring Variable Name
2. Declaring Class Name
3. Declaring Function Name
4. Declaring Object Name

ii) Identifier

Various data items with symbolic names in C++ is called as Identifiers. Following data items are called as Identifier in C++ –

1. Names of functions
2. Names of arrays
3. Names of variables
4. Names of classes

The rules of naming identifiers in C++ :

1. C++ is case-sensitive so that Uppercase Letters and Lower Case letters are different
2. The name of identifier cannot begin with a digit. However, Underscore can be used as first character while declaring the identifier.
3. Only alphabetic characters, digits and underscore (_) are permitted in C++ language for declaring identifier.
4. Other special characters are not allowed for naming a variable / identifier
5. Keywords cannot be used as Identifier.
   - Some valid examples:- sum, a1, a2, _1, _a, average, a_b, x123y...
   - Some invalid examples:- 1a, a-b, float

iii) Constants

A quantity or number that does not vary during the execution of a program is known as constant. Constants in C++ are categorized into three types,

(i) Numeric constants

(ii) Character constants

(iii) Symbolic constants.

(i) Numeric Constants: Numeric constants are categorized into integer constants and real constants.

(ii) Character Constant: Character constants are categorized into single character constants and string constants.

(iii) Symbolic Constants: Symbolic constants are similar to variables but their values cannot be modified after their initialization. They are categorized into three types - #define preprocessor directive, const keyword and enum keyword.

iv) Operators

Operators are the tokens that can join individual constants, variables, array elements and function references together. They act upon data items called operands. Following are some of the operators in C++.

+	<	&&	&	: :	comma
–	< =	||	|	:	sizeof
*	>	!	^	–>*	member selection
/	> =	+ +	< <	.*	new
%	= =	– –	> >		delete
	! =		–		

Table: C++ Operators

v) String Constants

String constants are the sequence of characters enclosed within double quotes.

Examples

"How are you"

"Book"

"Unjimited".

1.1.4 Data Typess

Q6. What is Data type? Explain briefly about primitive data type.

Ans :

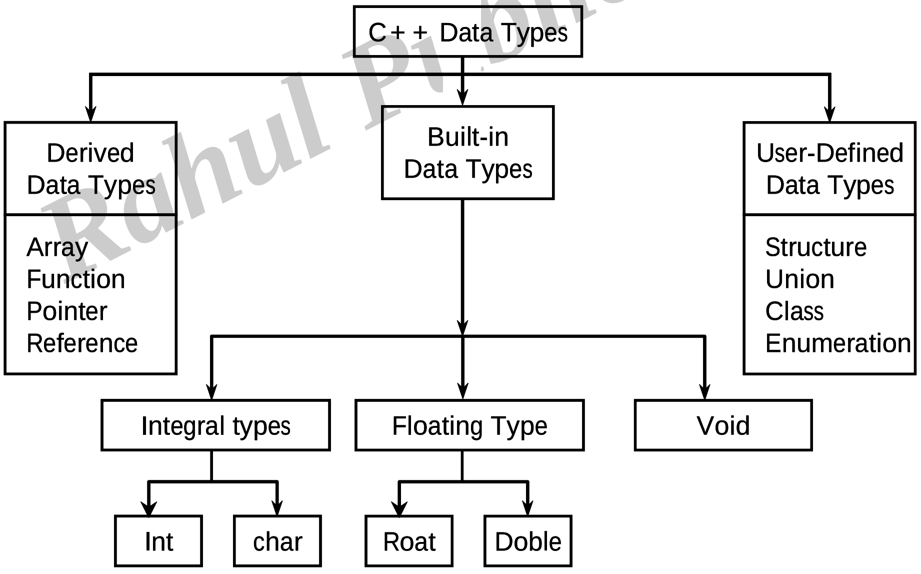
Data Types

A data type is a set of values and a set of predefined operations on those values. Every programming language deals with some data. For example to print some message or to solve some mathematical expression some data is necessary.

C++ is very rich in data types.

We can broadly divide all data type in c++ into three categories.

1. Primitive / Built in data types
2. Derived Data types
3. User defined Data types



Various Data Types in C++

Primitive Data Types

The primitive data types in c language are the inbuilt data types. Programmers can use these data types when creating variables in their programs.

For example, a programmer may create a variable called "rollno" and define it as a integer data type. The variable will then store data as a integer values.

The primitive data types can be classified into three general categories.

- Void
- Integral
- Floating –point

Void Type

The void type is denoted by the keyword void, it means no values and no operations. Void can be mostly used to designate that a function has no parameters and no return value and also define a pointer to generic data.

For example - void msg (void); -

For example - int average(void);-

Integral Type

Integral data type cannot contain a fractional part, they are whole numbers.

C language has three integral types.

1. Boolean
2. Character
3. Integer Type

1. Boolean

With release C99 ,C language incorporated a new data type, which is named after George Boole , who defined Boolean algebra. A variable of the boolean can have two values: true (1) and false (0).

The Boolean type is referred to the keyword bool.

Declaring the boolean : to declare boolean data type following syntax should be followed:

Syntax : *bool* variable_name = true / false ;

Example

```
bool is Pass;
bool is Done = false;
bool is TurnedOn = true;
```

Note : The header stdbool.h must be used to use Boolean data type.

Let us look at the following example

Example

```
#include <stdbool.h>
int main()
{
    bool gender[2] = {true, false};
    return 0;
}
```

Example

```
//Program to print Boolean constants
#include <stdbool.h>
int main()
{
// local declaration
    bool a = true;
    bool b= false;
cout<< "The Boolean values are :%d,
%d\n", a,b;
}
```

It gives the following output:

```
The Boolean values are : 1 0
```

2. Character

A character is any value that can be represented in the computer's alphabet, known as character set. char is a special integer type designed for storing single characters. The integer value of a char corresponds to an ASCII character.

Example

A value of 65 corresponds to the letter A, 66 corresponds to B, 67 to C, and so on.

The C standard provides two character types.

i) Char

ii) Wchar_t

i) Char: The key word char is used to represent character, which can store only one character. The size of char is 1 byte.

Declaring the char : to declare char data type following syntax should be followed:

Syntax: char variable_name = 'value';

Example : char c1= 'a'; (The value stored in ' ' represent the value in ascii)

Here, *c1* is a variable of type character which is storing a character 'a'.

For example

For, 'a', value=97

For, 'A', value=65

For, '2', value=49 (if enclosed in ' ' even the integer value is also converted into ASCII value)

For, '&', value=33

Example

```
int main()
{
char a, b, c;
a='E';
b='I';
c='O';
cout << "value of a= %c," a;
cout << "value of b= %c," b;
cout << "value of c= %c," c;
return 0;
}
```

The following program will give the output :

```
Value of a =E
Value of b =I
Value of c =O
```

ii) Wchar_t : Can hold a wide character. It is also required for declaring or referencing wide characters and wide strings.

wchar.h is a header file in the C standard library to represent wide characters.

*****NOTE:** It is beyond the scope of introductory programming.

3. Integer Type

Integer data type is used to declare a variable that can store numbers without a decimal. They can be in various sizes and sign, depending on whether they will take up the negative numbers or not. Type qualifiers are used to specify the sizes ans signs of the variable.

Declaring the integer : to declare integer data type following syntax should be followed:

Syntax : [*qualifier] int variab*

le_name; /* qualifier is optional*/

Keyword int is used for declaring the variable with integer type. The type of qualifier is used to represent size and sign qualifiers.

C++ supports four different type of integer data types:

- byte
- Short
- Int
- Long

S.No.	Data type	Size (in bytes)	range
1	byte	1	*127 to –128
2	short	2	+32767 to –32767
3	int	4	+2147483647 to –2147483647
4	long	8	+9223372036854775807 to –9223372036854775808

Floating Point Data types: C++ supports three floating point types. Real, imaginary and complex. Floating point types are always signed.

Real: Real type can hold the values with fractions. C supports three different sizes of real type, float, double and long double

Here, the details of the real type with storage sizes and value ranges and their precision.

Type	Storage size	Value range	Precision
Float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
Double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Declaring the real: to declare real data type following syntax should be followed:

Syntax: type variable = value

For example

```
float f1;
double f2;
```

Here, both f1 and f2 are floating type variables.

In C++, floating values can be represented in exponential form as well.

For example: float f3=43.345e2

Example:

```
main()
{
    float age;
    age = 10.5;
    cout<< "The boy is over %f years old.\n", age;
}
```

It gives the following output :

```
The boy is over 10.5 years old
```

Imaginary

An imaginary number is the real number multiplied by square root of -1. Like real , an imaginary type can be three different sizes, float imaginary, double imaginary and long double imaginary.

Complex

A complex number is a combination of both real and imaginary number. It can be in three sizes. float complex, double complex and long double complex.

NOTE: Complex data types use the header file complex.h

Declaring the complex : to declare integer data type following syntax should be followed:

Syntax : type variable = value;

Example : double complex a;

Q7. Write a program to demonstrate complex data type.

Ans :

```
#include <math.h>
#include <complex.h>
int main()
{
//  variable declaration
    double complex x = 3 +4 *I;
    double complex y = 3 -4 *I;
    double complex sum;
//  statements
sum = x + y;
cout<< "%f %f %f %f\n", creal(sum), cimaginary(sum), cabs(sum), carg(sum);
return 0;
}
```

It gives the following output :

```
6.000000 0.000000 6.000000 0.000000
```

Q8. What are various derived and user defined data types in C++.

Ans :

(A) Derived Data Types

Data types that are derived from the built-in data types are known as derived data types. The various derived data types provided by C++ are arrays, junctions, references and pointers.

i) Array

An array is a set of elements of the same data type that are referred to by the same name. All the elements in an array are stored at contiguous (one after another) memory locations and each element is accessed by a unique index or subscript value. The subscript value indicates the position of an element in an array.

ii) Function

A function is a self-contained program segment that carries out a specific well-defined task. In C++, every program contains one or more functions which can be invoked from other parts of a program, if required.

iii) Pointer

A pointer is a variable that can store the memory address of another variable. Pointers allow to use the memory dynamically. That is, with the help of pointers, memory can be allocated or de-allocated to the variables at run-time, thus, making a program more efficient.

(B) User-Defined Data Types

Various user-defined data types provided by C++ are structures, unions, enumerations and classes.

i) Structure, Union and Class

Structure and union are the significant features of C language. Structure and union provide a way to group similar or dissimilar data types referred to by a single name. However, C++ has extended the concept of structure and union by incorporating some new features in these data types to support object -oriented programming.

C++ offers a new user-defined data type known as class, which forms the basis of object-oriented programming. A class acts as a template which defines the data and functions that are included in an object of a class. Classes are declared using the keyword class. Once a class has been declared, its object can be easily created.

ii) Enumeration

An enumeration is a set of named integer constants that specify all the permissible values that can be assigned to enumeration variables. These set of permissible values are known as enumerators. For example, consider this statement.

```
enum country {US, UN, India, China};
    // declaring an
    // enum type
```

In this statement, an enumeration data-type country (country is a tag name), consisting of enumerators US, UN and so on, is declared. Note that these enumerators represent integer values, so any arithmetic operation can be performed on them.

1.1.5 Variables

Q9. What is variable? Briefly explain about how to use variables.

Ans :

Variable is a location in the computer memory which can store data and is given a symbolic name for easy reference. The variables can be used to hold different values at different times during the execution of a program.

Basic types of Variables

Each variable while declaration must be given a datatype, on which the memory assigned to the variable depends. Following are the basic types of variables,

- bool For variable to store boolean values (True or False)
- char For variables to store character types.
- int for variable with integral values
- float and double are also types for variables with large and floating point values

Declaration and Initialization

Variable must be declared before they are used. Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

Example

```
int i;              // declared but not initialised
char c;
int i, j, k;        // Multiple declaration
```

Initialization means assigning value to an already declared variable,

```
    int i;          // declaration
    i = 10;         // initialization
```

Initialization and declaration can be done in one single step also,

```
    int i=10;       // initialization and declaration in same step
    int i=10, j=11;
```

If a variable is declared and not initialized by default it will hold a garbage value. Also, if a variable is once declared and if try to declare it again, we will get a compile time error.

```
int i,j;
i=10;
j=20;
int j=i+j;          // compile time error, cannot redeclare a variable in same scope
```

Q10. What is reference variable in C++.

Ans :

Reference variable

Reference variable is a new feature added to C++. A reference variable basically assigns an alternative name to an existing variable.

The syntax for creating reference variable is :

Data_type & reference_name = variable_name

For ex: int n=10;

Int &num = nl

Then both n and num are the integer type variables and will store the same data.

1.1.6 Operators

Q11. Describe the various operators in C++.

Ans :

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators.

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Miscellaneous Operators

1. Arithmetic Operators

There are following arithmetic operators supported by C++ language “

Assume variable A holds 10 and variable B holds 20, then “

Examples

Operator	**Description**	**Example**
+	Adds two operands	A + B will give 30
–	Subtracts second operand from the first	A – B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

2. Relational Operators

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then H

Examples

Operator	**Description**	**Example**
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

3. Logical Operators

There are following logical operators supported by C++ language.

Assume variable A holds 1 and variable B holds 0, then H

Examples

Operator	**Description**	**Example**
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then Condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

4. Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows H

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows “

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A ^ B = 0011 0001

~A = 1100 0011

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then “

Examples

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

5. Assignment Operators

There are following assignment operators supported by C++ language H

Examples

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and Assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^ =	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

6. Miscellaneous Operators

The following table lists some other operators that C++ supports.

Sr.No	Operator & Description
1	**sizeof** sizeof operator returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return 4.
2	**Condition ? X : Y** Conditional operator (?). If Condition is true then it returns value of X otherwise returns value of Y.
3	Comma Operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
4	**(dot) and -> (arrow)** Member operators are used to reference individual members of classes, structures, and unions.
5	**Cast** Casting operators convert one data type to another. For example, int(2.2000) would return 2.
6	**&** Pointer operator & returns the address of a variable. For example &a; will give actual address of the variable.
7	***** Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var.

Operators Precedence in C++

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator H

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Examples

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

1.1.7 Expressions

Q12. Explain various Expressions in C++?

Ans :

A combination of variables, constants and operators that represents a computation forms an expression. Depending upon the type of operands involved in an expression or the result obtained after evaluating expression,

i) **Constant expressions:** The expressions that comprise only constant values are called constant expressions. Some examples of constant expressions are 20, 'a' and 2/5+30 .

ii) **Integral expressions:** The expressions that produce an integer value as output after performing all types of conversions are called integral expressions. For example, x, 6*x-y and 10 +int (5.0) are integral expressions. Here, x and yare variables of type into.

iii) **Float expressions:** The expressions that produce floating-point value as output after performing all types of conversions are called float expressions. For example, 9.25, x-y and 9+ float (7) are float expressions. Here, x 'and yare variables of type float.

iv) **Relational or Boolean expressions:** The expressions that produce a bool type value, that is, either true or false are called relational or Boolean expressions. For example, x + y<100, m + n==a-b and a>=b + c .are relational expressions.

v) **Logical expressions:** The expressions that produce a bool type value after combining two or more relational expressions are called logical expressions. For example, x==5 &&m==5 and y>x, m<=n are logical expressions.

vi) **Bitwise expressions:** The expressions which manipulate data at bit level are called bitwise expressions. For example, a >> 4 and b<< 2 are bitwise expressions.

vii) **Pointer expressions:** The expressions that give address values as output are called pointerex-pressions. For example, &x, ptr and -ptr are pointer expressions. Here, x is a variable of any type and ptr is a pointer.

viii) **Special assignment expressions:** An expression can be categorized further depending upon the way the values are assigned to the variables.

ix) **Chained assignment:** Chained assignment is an assignment expression in which the same value is assigned to more than one variable, using a single statement. For example, consider these statements.

```
a = (b=20); or a=b=20;
```

In these statements, value 20 is assigned to variable b and then to variable a. Note that variables cannot be initialized at the time of declaration using chained assignment. For example, consider these statements.

```
int a=b=30; // illegal
int a=30, int b=30; //valid
```

x) **Compound Assignment**

Compound Assignment is an assignment expression, which uses a compound assignment operator that is a combination of the assignment operator with a binary arithmetic operator. For example, consider this statement.

```
a + = 20; //equivalent to a=a+20;
```

In this statement, the operator += is a compound assignment operator, also known as short-hand assignment operator.

Q13. Write a program to illustrate relational and logical expressions.

Ans :

```
#include<iostream.h>
# include<conio.h>
int main()
{
int x,y;
clrscr();
cout < < “Enter the value of x:”;
cin > > x;
cout < < “Enter the value of y:”;
cin > > y;
cout < < “x > y is” < <(x > y)< < endl;
cout < < “x < y is” < <(x < y)< < endl;
cout < <”x > = y is” < <(x > = y)< < endl;
cout < < “x < = y is” < <(x < = y)< < endl;
cout < < “x = y is” < <(x = = y)< < endl;
cout < < “x != y is” < < (x != y) < < endl;
cout < <“x && y is” < <(x && y)< < endl;
cout < < “x || y is” < <(x || y)< < endl;
cout < < “(x < 10) && (y < 10) is”
< < ((x < 10) && (y < 10)) < < endl;
getch();
return 0;
}
```

Output

```
DOS
BOX        DOSBox 0.74, Cpu speed: max 100%

 Enter the value of × : 5
 Enter the value of y : 6
 × > y is 0
 × < y is 1
 × > = y is 0
 × < = y is 1
 × = = y is 0
 × ! = y is 1
 × && y is 1
 × ¦¦ y is 1
  (× < 10) && (y < 10) is 1
```

1.1.8 Control Structures

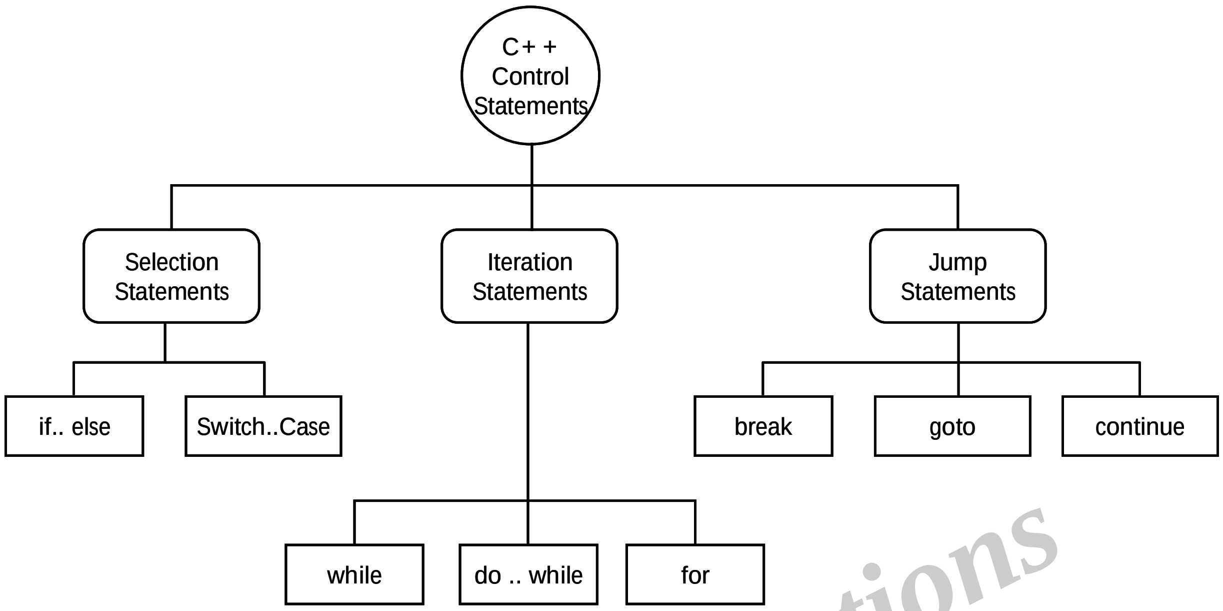
Q14. Explain briefly about Control Structures.

Ans :

Control flow or flow of control is the order in which instructions, statements and function calls being executed or evaluated when a program is running. The control flow statements are also called as Flow Control Statements. In C++, statements inside your code are generally executed sequentially from top to bottom, in the order that they appear. It is not always the case your program statements to be executed straightforward one after another sequentially, you may require to execute or skip certain set of instructions based on condition, jump to another statements, or execute a set of statements repeatedly. In C++, control flow statements are used to alter, redirect, or to control the flow of program execution based on the application logic.

C++ Control Flow Statement Types

In C++, Control flow statements are mainly categorized in following types -



C++ Selection Statements

In C++, Selection statements allow you to control the flow of the program during run time on the basis of the outcome of an expression or state of a variable. Selection statements are also referred to as Decision making statements. Selection statements evaluates single or multiple test expressions which results in 'TRUE" or "FALSE". The outcome of the test expression/condition helps to determine which block of statement(s) to executed if the condition is'TRUE" or "FALSE" otherwise.

In C++, we have following selection statements -

C++ Jump Statements

Jump statements are used to alter or transfer the control to other section

or statements in your program from the current section.

In C++, we have following types of jump statements -

- C++ Break Statement
- C++Continue Statement
- C++ goto Statement

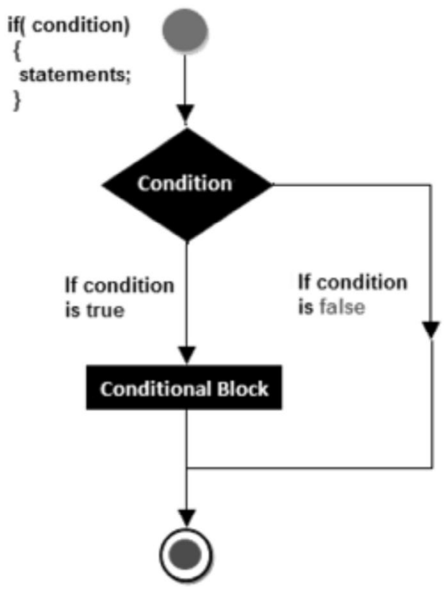
All of the above jump statements cause different types of jumps.

Q15. Explain If statement in C++ with syntax and example.

Ans :

The if statement checks whether the test condition is true or not. If the test condition is true, it executes the code/s inside the body of if statement. But it the test condition is false, it skips the code/s inside the body of if statement.

The if keyword is followed by test condition inside parenthesis ( ). If the test condition is true, the codes inside curly bracket is executed but if test condition is false, the codes inside curly bracket { } is skipped and control of program goes just below the body of if as shown in figure above.



Syntax: If(condition)

```
        {
        Statements
        ..........
        }
```

Example

program to find the maximum of two numbers

```
#include< iostream.h>
int main( )
{
 int x,y;
 x=15;
 y=13;
 if (x > y )
 {
  cout << "x is greater than y";
 }
}
```

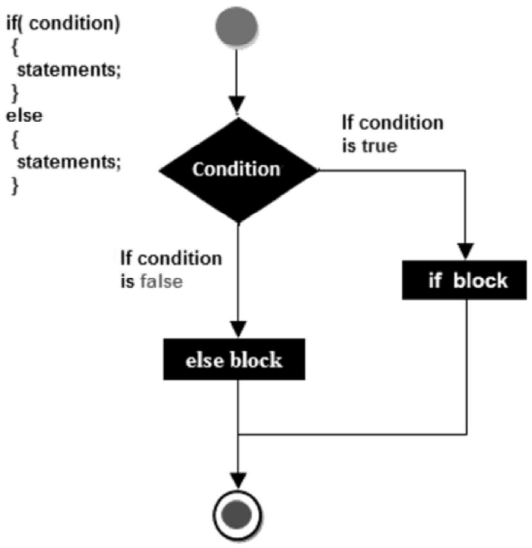
Output :

```
x is greater than y
```

Q16. Explain If else statement with syntax and example.

Ans :

In general it can be used to execute one block of statement among two blocks:



Syntax:

```
if( expression )
{
 statement-block1;
}
else
{
 statement-block2;
}
```

If the 'expression' is true, the 'statement-block1' is executed, else 'statement-block1' is skipped and 'statement-block2' is executed.

Example

Program to find greater than of two numbers using if – else

```
void main( )
{
 int x,y;
 x=15;
 y=18;
 if (x > y )
 {
```

```
 cout < < "x is greater than y";
 }
else
 {
 cout < < "y is greater than x";
 }
}
```

Output :

```
y is greater than x
```

Q17. Explain Nested if...else statement with syntax and example.

Ans :

Nested if...else are used if there are more than one test expression.

Syntax:

```
if( expression )
{
  if( expression1 )
   {
     statement-block1;
   }
  else
   {
     statement-block2;
   }
}
else
{
 statement-block3;
}
```

if 'expression' is false the 'statement-block3' will be executed, otherwise it continues to perform the test for 'expression 1' . If the 'expression 1' is true the 'statement-block1' is executed otherwise 'statement-block2' is executed.

Example

program to find the greatest of three numbers

```
void main( )
{
 int a,b,c;
 clrscr();
 cout < < "enter 3 number";
 cin > > a > > b > > c;
 if(a > b)
 {
  if( a > c)
  {
   cout < < "a is greatest";
  }
  else
  {
   cout < < "c is greatest";
  }    }
 else
 {
  if( b> c)
  {
   cout < < "b is greatest";
  }
  else
  {
   printf("c is greatest");
  }    }
getch();
}
```

Q18. Write the syntax of else-if ladder.

Ans :

Syntax:

```
if(expression 1)
{
 statement-block1;
}
```

```
else if(expression 2)
{
 statement-block2;
}
else if(expression 3 )
{
 statement-block3;
}
else
        default-statement;
```

The expression is tested from the top(of the ladder) downwards. As soon as the true condition is found, the statement associated with it is executed.

Q19. Write a Program to find whether the given number is divisible by 5 or 8.

Ans :

```
void main( )
{
 int a;
 cout << "enter a number";
 cin >> a;
 if( a%5==0 && a%8==0)
 {
  cout << "divisible by both 5 and 8";
 }
 else if( a%8==0 )
 {
  cout << "divisible by 8";
 }
 else if(a%5==0)
 {
  cout << "divisible by 5";
 }
 else
 {
  cout << "divisible by none";
 }
getch();
}
```

OUTPUT

```
Enter a number : 40
Divisible by both 5 and 8
```

Q20. Write a program to Check Vowel or a Consonant Manually

Ans :

```
#include<iostream>
usingnamespace std;
int main()
{
char c;
int isLowercaseVowel, isUppercaseVowel;
    cout <<"Enter an alphabet: ";
    cin >> c;
        // evaluates to 1 (true) if c is a lowercase vowel
        isLowercaseVowel =(c =='a'|| c =='e'|| c
        =='i'|| c =='o'|| c =='u');
        // evaluates to 1 (true) if c is an uppercase
           vowel
        isUppercaseVowel =(c =='A'|| c =='E'||
        c =='I'|| c =='O'|| c =='U');
        // evaluates to 1 (true) if either
        // isLowercaseVowel
         //or  isUppercaseVowel is true
        if(isLowercaseVowel || isUppercaseVowel)
        cout << c <<" is a vowel.";
        else
        cout << c <<" is a consonant.";
        return0;
        }
```

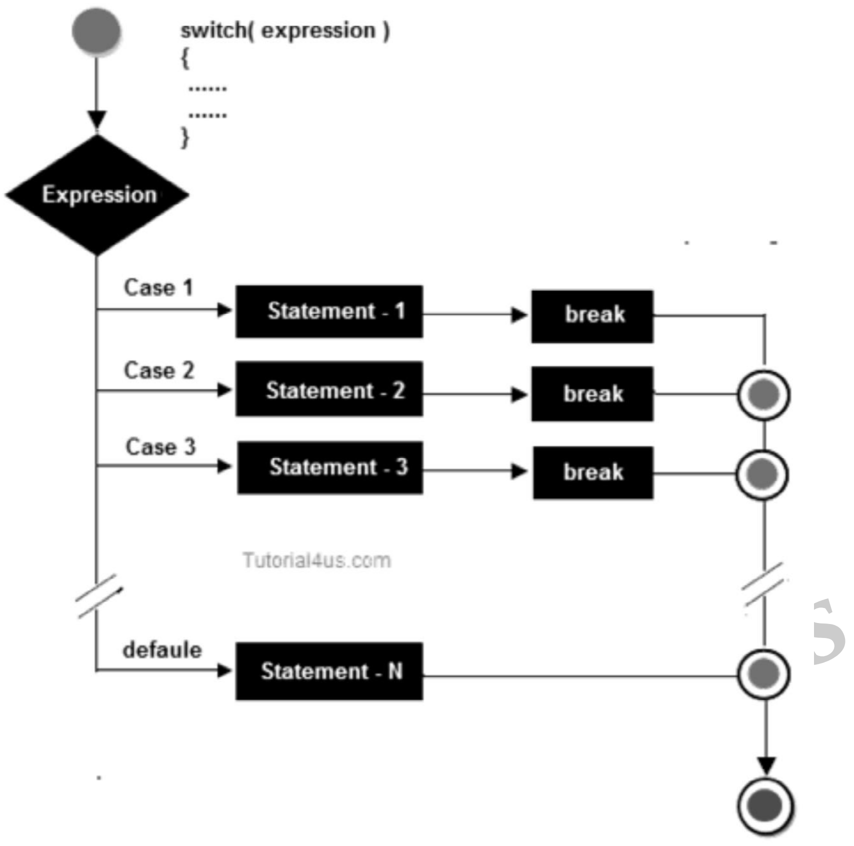
Output

```
Enter an alphabet: u
u is a vowel.
```

Q21. Write the syntax of Switch statement. Explain it with example.

Ans :

A switch statement work with byte, short, char and int primitive data type, it also works with enumerated types and string.



Rules for Apply Switch

1. With switch statement use only byte, short, int, char data type.
2. You can use any number of case statements within a switch.
3. Value for a case must be same as the variable in switch .

Limitations of Switch

Logical operators cannot be used with switch statement. For instance

Example

case k>=20: //is not allowed

Switch case variables can have only int and char data type. So float data type is not allowed.

Syntax : switch(ch)

```
{
case1:
statement 1;
break;
case2:
statement 2;
break;          }
```

In this ch can be integer or char and cannot be float or any other data type.

Example of Switch case

```
#include<iostream.h>
#include<conio.h>
void main()
{
int ch;
clrscr();
cout<<"Enter any number (1 to 7)";
cin>>ch;
switch(ch)
{
case 1:
cout<<"Today is Monday";
break;
case 2:
cout<<"Today is Tuesday";
break;
case 3:
cout<<"Today is Wednesday";
break;
case 4:
cout<<"Today is Thursday";
break;
case 5:
cout<<"Today is Friday";
break;
case 6:
cout<<"Today is Saturday";
break;
case 7:
cout<<"Today is Sunday";
break;
default:
cout<<"Only enter value 1 to 7";
}
getch();
}
```

OUTPUT

```
Enter any number (1 to 7): 5

Today is Friday
```

*****Note:** In switch statement default is optional but when we use this in switch default is executed at last whenever all cases are not satisfied the condition.

Q22. Write a program to print Simple Calculator using switch statement.

Ans :

```
# include <iostream>
usingnamespace std;
int main()
{
char op;
float num1, num2;
   cout <<"Enter operator either + or - or *
             or /: ";
   cin >> op;
   cout <<"Enter two operands: ";
   cin >> num1 >> num2;
switch(op)
{
case'+':
         cout << num1+num2;
break;

case'-':
         cout << num1-num2;
break;

case'*':
         cout << num1*num2;
break;

case'/':
         cout << num1/num2;
break;
default:
       // If the operator is other than +, -, * or
       //, error message is shown
       cout <<"Error! operator is not correct";
break;
}
return0;
}
```

Output

```
Enter operator either + or - or * or divide : -
Enter two operands:
3.4
8.4
3.4 - 8.4 = -5.0
```

Q23. What are iterative statements? What are its types?

Ans :

Iterative statements / Loops

In any programming language, loops are used to execute a set of statements repeatedly until a particular condition is satisfied.

A sequence of statement is executed until a specified condition is true. This sequence of statement to be executed is kept inside the curly braces { } known as loop body. After every execution of loop body, condition is checked, and if it is found to be true the loop body is executed again. When condition check comes out to be false, the loop body will not be executed.

There are 3 type of loops in C++ language

1. while loop
2. for loop
3. do-while loop

Q24. Explain while loop with syntax and example.

Ans :

while loop can be address as an entry control loop. It is completed in 3 steps. In while loop First check the condition if condition is true then control goes inside the loop body other wise goes outside the body. while loop will be repeats in clock wise direction.

- ➢ Variable initialization.(e.g int x=0;)
- ➢ condition(e.g while(x<=10))
- ➢ Variable increment or decrement (x++ or x— or x=x+2)

Syntax

```
variable initialization ;
while (condition)
{
 statements ;
 variable increment or decrement ;
}
```

*****Note:** If while loop condition never false then loop become infinite loop.

Example : print the natural numbers below 5

```
#include<iostream.h>
#include<conio.h>
void main()
{
int i;
clrscr();
i=1;
while(i<5)
{
cout<<endl<<i;
i++;
}
getch();
}
```

OUTPUT

```
1
2
3
4
```

Q25. Explain for loop with syntax and example.

Ans :

for loop is used to execute a set of statement repeatedly until a particular condition is satisfied.

For loop contains 3 parts.

- ➢ Initialization
- ➢ Condition
- ➢ Iteration

Syntax:

```
for(initialization; condition ; increment/decrement)
{
  statement-block;
}
```

- When we are working with for loop always execution process will start from initialization block.
- After initialization block control will pass to condition block, if condition is evaluated as true then control will pass to statement block.
- After execution of the statement block control will pass to iteration block, from iteration it will pass back to the condition.
- Always repetitions will happen beginning condition, statement block and iteration only.
- Initialization block will be executed only once when we are entering into the loop first time.
- When we are working with for loop everything is optional but mandatory to place 2 semicolons (; ;)

Example

```
while()          // Error
for( ; ; )       // valid
```

Example : print the natural numbers below 5 using for loop

```
#include<iostream.h>
#include<conio.h>
void main()
{
int i;
clrscr();
for(i=1;i<5;i++)
{
cout<<endl<<i;
}
getch();
}
```

OUTPUT

```
1
2
3
4
```

Q26. Write a program, to find the given number is prime or not using for loop.

Ans :

```
#include<iostream>
usingnamespace std;
int main()
{
int n, i;
bool isPrime = true;
  cout <<"Enter a positive integer: ";
  cin >> n;
for(i = 2; i <= n / 2; ++i)
  {
if(n % i == 0)
      {
        isPrime = false;
break;
      }
  }
if (isPrime)
      cout <<"This is a prime number";
else
      cout <<"This is not a prime number";
return0;
}
```

Output

```
Enter a positive integer: 29
This is a prime number.
Enter a positive integer: 12
This is not a prime number.
```

Q27. Explain do while loop with syntax and example.

Ans :

In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of do-while loop. do statement evaluates the body of the loop first and at the end, the condition is checked using while statement. General format of do-while loop is,

Syntax

```
do
{
 ....
 .....
}
while(condition);
```

When use Do-While Loop

When we need to repeat the statement block atleast 1 time then we use do-while loop.

Example

program to print the natural numbers below 5

```
#include<iostream.h>
#include<conio.h>
void main()
{
int i;
clrscr();
i=1;
do
{
cout<<endl<<i;
i++;
}
while(i<5);
getch();
}
```

OUTPUT

```
1
2
3
4
```

Q28. Write a program to find GCD of a number using while loop.

Ans :

```
#include<iostream>
usingnamespace std;
int main()
{
int n1, n2;
   cout <<"Enter two numbers: ";
   cin >> n1 >> n2;

while(n1 != n2)
{
if(n1 > n2)
          n1 -= n2;
else
          n2 -= n1;
}

   cout <<"HCF = "<< n1;
return0;
}
```

Output

```
Enter two numbers: 78
52
HCF = 26
```

Q29. What is Nested Loop? Explain the syntax of nested loop with an example.

Ans :

In Nested loop one loop is place within another loop body.

When we need to repeated loop body itself n number of times use nested loops. Nested loops can be design upto 255 blocks.

Nested for loop

We can also have nested for loop, i.e one for loop inside another for loop. Basic syntax is,

```
for(initialization; condition; increment/decrement)
{
  for(initialization; condition; increment/decrement)
    {
      statement ;
    }
}
```

nested while loop

The syntax of nested while loop is as follows

```
while(condition)
{
   while(condition)
```

```
  {
     statement(s);
  }
  statement(s); // you can put more statements.
}
```

nested do...while loop

The syntax of nested do loop is as follows

```
do
{
  statement(s); // you can put more statements.
  do
  {
     statement(s);
  }while( condition );

}while( condition );
```

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100:

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

  for(i=2; i<50; i++) {
     for(j=2; j <= (i/j); j++)
       if(!(i%j)) break; // if factor found, not prime
       if(j > (i/j)) cout << i << " is prime\n";
  }
  return 0;
}
```

This would produce the following result:

OUTPUT

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
```

Q30. Define Jumping out of loop.

Ans :

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becocmes true, that is jump out of loop. C language allows jumping from one statement to another within a loop as well as jumping out of the loop.

Q31. Write about the following statements with syntax and examples.

i) Break statement

ii) Continue statement

iii) Goto statment

Ans :

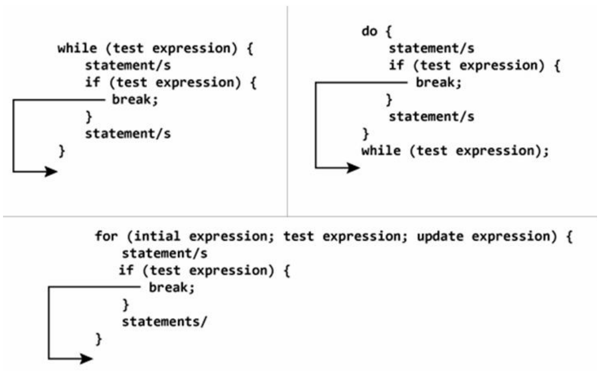
i) Break Statement

When break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

Syntax of break : break;

In real practice, break statement is almost always used inside the body of conditional statement (if...else) inside the loop.

Working of break statement



```
while (test expression) {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
```

```
do {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
while (test expression);
```

```
for (intial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        break;
    }
    statements/
}
```

*****NOTE :** In C programming, break statement is also used with switch...case statement.

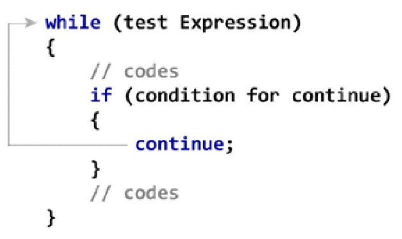
ii) Continue statement

It causes the control to go directly to the test-condition and then continue the loop process. On encountering continue, cursor leave the current cycle of loop, and starts with the next cycle.

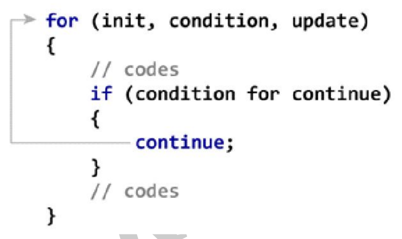
The continue statement skips some statements inside the loop. The continue statement is used with decision making statement such as if...else.

Syntax of continue Statement : continue;

How continue statement works?



```
while (test Expression)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```



```
for (init, condition, update)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

iii) Go to statement

In C++ programming, goto statement is used for altering the normal sequence of program execution by transferring control to some other part of the program.

Syntax of goto Statement : goto label;

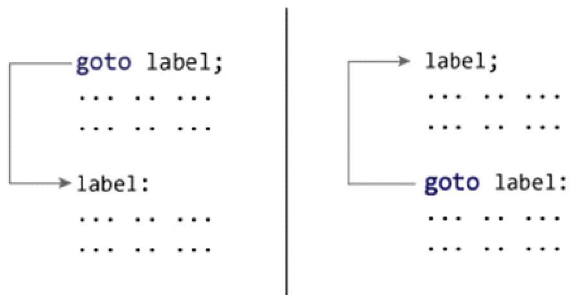
... .

label:

statement;

... .

In syntax above, label is an identifier. When goto label; is encountered, the control of program jumps to label: and executes the code below it.



```
goto label;                 label;
... .. ...                  ... .. ...
... .. ...                  ... .. ...

label:                      goto label:
... .. ...                  ... .. ...
... .. ...                  ... .. ...
```

Example : goto Statement

```
        /* C++ program to demonstrate the
           working of goto statement. */
        /* This program calculates the average of
           numbers entered by user. */
        /* If user enters negative number, it ignores
           that number and calculates the average
           of number entered before it.*/
# include <iostream>
using namespace std;
int main() {
  float num, average, sum = 0.0;
  int i, n;
  cout<<"Maximum number of inputs: ";
  cin>>n;

  for(i=1; i <= n; ++i) {
    cout<<"Enter n"<<i<<": ";
    cin>>num;

    if(num < 0.0) {
      goto jump;  /* Control of the program
                  / moves to jump; */
    }
    sum += num;
  }
jump:
  average=sum/(i-1);
  cout<<"\nAverage = "<<average;
  return 0;
}
```

Output

```
Maximum number of inputs: 10
Enter n1: 2.3
Enter n2: 5.6
Enter n3: -5.6
Average = 3.95
```

1.1.9 Arrays

Q32. Define Array How to declare and initialise an array in C++.

Ans :

An array is a collection of data that holds fixed number of values of same type. For example:

int age[100];

Here, the *age* array can hold maximum of 100 elements of integer type.

The size and type of arrays cannot be changed after its declaration.

declaring an array

dataType arrayName[arraySize];

For example,

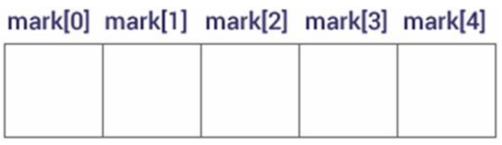
float mark[5];

Here, we declared an array, *mark*, of floating-point type and size 5. Meaning, it can hold 5 floating-point values.

Elements of an Array

You can access elements of an array by using indices.

Suppose you declared an array mark as above. The first element is mark[0], second element is mark[1] and so on.



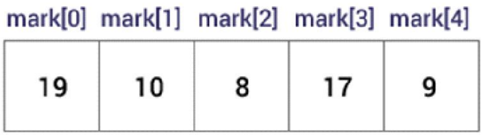
Initializing an array :

It's possible to initialize an array during declaration. For example,

int mark[5] = {19, 10, 8, 17, 9};

Another method to initialize array during declaration:

int mark[] = {19, 10, 8, 17, 9};



Here,

mark[0] is equal to 19

mark[1] is equal to 10

mark[2] is equal to 8

mark[3] is equal to 17

mark[4] is equal to 9

Q33. Define one dimensional Arrary. What are the differences between one and two dimensional array?

Ans :

One-dimensional array is an array that stores elements in contiguous memory locations and accesses them using only one subscript.

Syntax

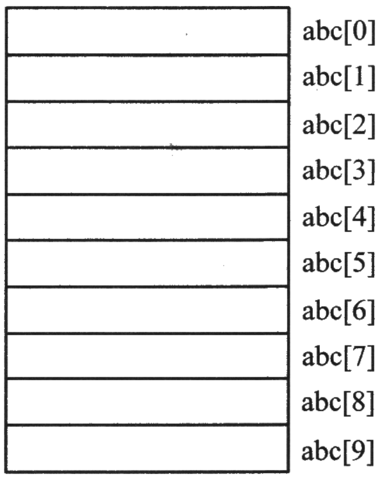
datatype array_name[size];

Example

int abc[10];

Here, abc is an array of 10 integers.

This means, the computer reserves ten storage locations in the memory as,



Differences between One and Two Dimensional Arrays

One-dimensional Array	Two-dimensional Array
1. It stores a set of elements of similar data type.	1. It stores 'list of lists', 'array of arrays' or 'array of one dimensional arrays.

2. **Syntax**
type array_name[size];
(or)
type array_name[ ];
array name = new type [size];
	2. Syntax
	type array_name[sizel] [size2];
	(or)
	type array_name = newtype [sizel] [size2];

3. It needs only one value to depict the size of array.	3. It needs two values to depict the size of the i.e., one for rows and array the other for columns.
4. Receiving parameter can be a pointer sized array or an unsized array.	4. Receiving parameter can be the right most dimension of the array.

5. Example
int array[5];
(or)
int array[ ];
array = new int [5];
	5. Example
	int array[2][3];
	(or)
	int array = new int[2] [3];

Q34. Write a program to insert and print array elements?

Ans :

```
int mark[5] = {19, 10, 8, 17, 9}
// insert different value to third element mark[3] = 9;
// take input from the user and insert in third element cin >> mark[2];
// take input from the user and insert in (i+1)th element
cin >> mark[i];
// print first element of an array
cout << mark[0];
// print ith element of an array
cin >> mark[i-1];
```

Q35. Write a C++ program to store and calculate the sum of 5 numbers entered by the user using arrays.

Ans :

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

```
{
    int numbers[5], sum = 0;
    cout << "Enter 5 numbers: ";
   // Storing 5 number entered by user in an array
    // Finding the sum of numbers entered
    for (int i = 0; i < 5; ++i)
    {
        cin >> numbers[i];
        sum += numbers[i];
    }
    cout << "Sum = " << sum << endl;
    return 0;
}
```

Output

```
Enter 5 numbers: 3
4
5
4
2
Sum = 18
```

Q36. Write a program to Display Largest Element of an array.

Ans :

```
#include<iostream>
usingnamespace std;
int main()
{
int i, n;
float arr[100];
     cout <<"Enter total number of elements(1 to 100):";
    cin >> n;
    cout << endl;
// Store number entered by the user
for(i =0; i < n;++i)
{
       cout <<"Enter Number "<< i +1<<" : ";
       cin >> arr[i];
}
// Loop to store largest number to arr[0]
for(i =1;i < n;++i)
{
// Change < to > if you want to find the smallest element
if(arr[0]< arr[i])
            arr[0]= arr[i];
}
    cout <<"Largest element = "<< arr[0];
return0;
}
```

Output

```
Enter total number of elements: 8
Enter Number 1: 23.4
Enter Number 2: -34.5
Enter Number 3: 50
Enter Number 4: 33.5
Enter Number 5: 55.5
Enter Number 6: 43.7
Enter Number 7: 5.7
Enter Number 8: -66.5
Largest element = 55.5
```

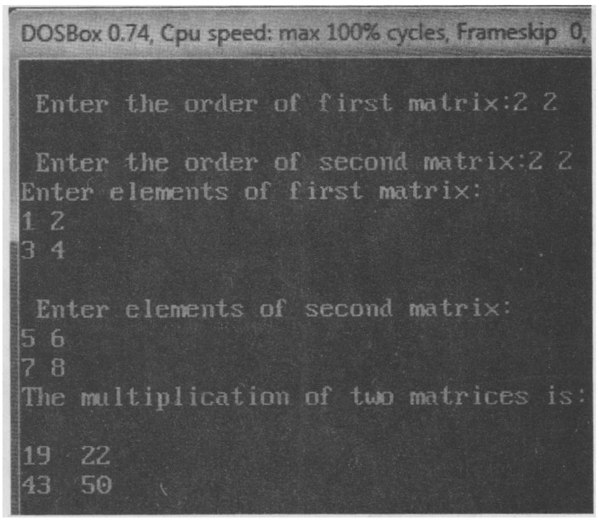
Q37. Write a C++ program to multiplication of two matrices.

Ans :

Program

```
#include<iostream.h>
#include<conio.h>
void main()
{
int rl, cl, r2, c2, i, j, k;
int a[10][10], b[10][10], c[10][10];
clrscr( );
cout<<"\n Enter the order of first matrix:";
cin>>rl>>c1;
cout<<"\n Enter the order of second matrix:";
cin>>r2>>c2;
cout<<"Enter elements of first matrix:";
for(i = 0; i<r1; i++)
{
for(j = 0; j<c1; j++)
{
cin>>a[i][j];
}
}
cout<<"\n Enter elements of second matrix:"; for(i
= 0; i< r2; i++)
{
for(j = 0; j< c2;j++)
{
cin>>b[i][j];
}
}/*Matrix multiplication */ if(cl = r2)
{
for(i = 0; i <r2; i++)
{
for(j = 0; j <c2; j++)
{
c[i][j] = 0;
for(k = 0; k<r2; k++)
c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
}
cout<<"The multiplication of two matrices is:\n";
for(i = 0; i <r1; i++)
{
cout<<end1; for(0 = 0; j <c1; j++)
{
cout<<c[i][j];
cout<<"  ";
}
}
}
else
cout<<"\n Multiplication not possible";
getch();
}
```

Output



```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0,

 Enter the order of first matrix:2 2

 Enter the order of second matrix:2 2
Enter elements of first matrix:
1 2
3 4

 Enter elements of second matrix:
5 6
7 8
The multiplication of two matrices is:

19  22
43  50
```

1.1.9.1 Searching and Sorting Arrays

Q38. Give an algorithm and explain the concept of linear or sequential search along with an example program.

Ans :

The sequential/linear search method is mainly applicable for searching an element within an unordered list. The principle behind sequential/linear search is that each element of the list is compared or matched with the key (i.e., search argument) in a sequential order. The search (i.e., comparison) begins from the starting element and proceeds until a match is found or the end of list is encountered. If a match is found (i.e., successful search), the position (i.e., index) of the matched element is returned. If a match is not found but the end of list is encountered then - 1 is returned (i.e., unsuccessful search).

Algorithm

Step 1

Initialize index (i.e., position) i = 0 and number of elements in list = n

Step 2

Check, if (i < n) then

Goto step 3

else

Goto step 5

Step 3

```
Compare, if (key matches with list element at i) then
cout<<"Element found ! successful search";
Return the index (i) of the found element
Goto step 6
else
```

Step 4

```
Increment i value by 1
Goto step 2
```

Step 5

```
Print"End of list; Element not found ! unsuccessful search";
```

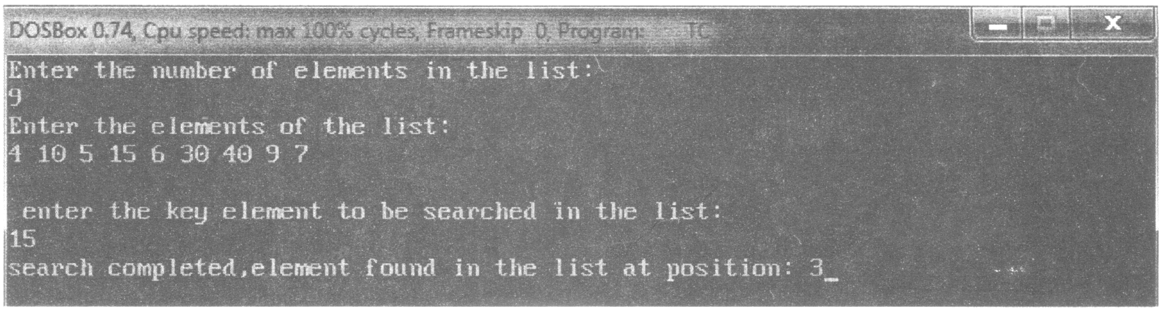
Step 6

```
Stop
```

Program

```
	# include<iostream.h>
	# include<conio.h>
	int linear_search(int [ ], int, int);
	int main()
{
	int n,A[20], i, key, index;
	clrscr();
	cout<< "Enter the number of elements in the list:\n";
	cin>>n;
	cout<<"Enter the elements of the list:\n";
	for(i=0; i<n; i++)
	cin>>A[i];
	cout<< "\n enter the key element to be searched in the list:\n";
	cin>>key;
	index=linear_search(A, n, key);
	if(index==-l)
	cout<<"search completed, element not found\n";
	else
	cout<<"search completed,element found in the list at position:''<<index;
	getch();
	return 0;
}
	int linear_search(int 1[ ], int n, int e)
{
        int i;
        for(i = 0; i<n; i++)
        if(l[i]=e)
        return i;
        return -1;
}
```

Output


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
Enter the number of elements in the list:
9
Enter the elements of the list:
4 10 5 15 6 30 40 9 7

 enter the key element to be searched in the list:
15
search completed,element found in the list at position: 3_
```

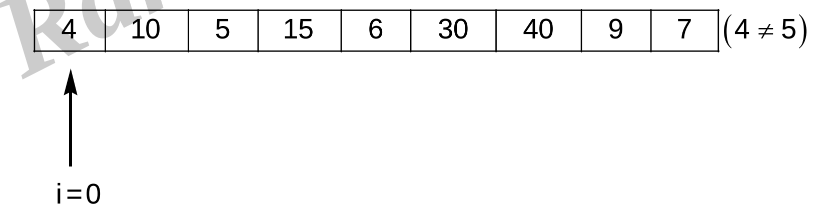
Example

Consider the following list,

4	10	5	15	6	30	40	9	7

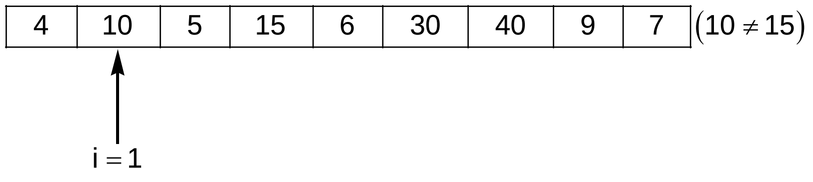
Let the element to be searched is '15'. The linear/sequential search algorithm starts from the first element i.e., 4 and compares it with the key element i.e., 15.

The index element is set to '0'.



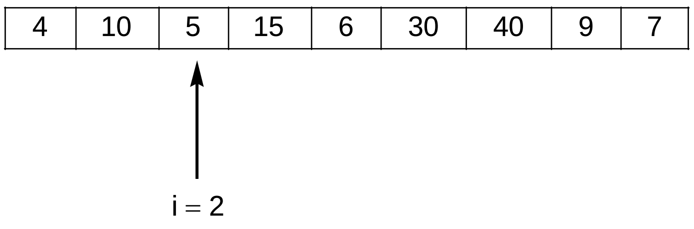
Since $4 \neq 15$, 'i' is moved to the next element i.e., 10.

Now, the element at i $=1$ is compared with the key element 15.



Since $10 \neq 15$, 't' is moved to the next element i.e., 5

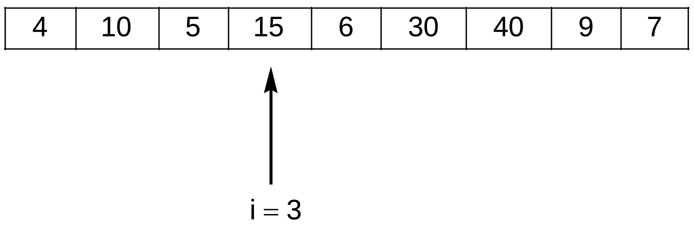
Now, the element at i = 2 is compared with the key element 15.



4	10	5	15	6	30	40	9	7

Since, $5 \neq 15$, 't' is moved to the next element i.e., 15

Now, the element i = 3 is compared with the key element '15.



4	10	5	15	6	30	40	9	7

Since, the current element matches with the key element (i.e., 15 = 15) at location i = 3, the search is successful and the location of element 't' i.e., 3 is returned upon success.

Time Complexity

Time complexity of linar search is O(n). It required atmost 'n' comparisons to search an element in a list of 'n' elements.

Q39. What is searching ? Explain the concept of binary search with an example program.

OR

Explain the concept of binary search with proper example and program.

Ans :

Searching

Searching is a process of finding the correct location of an element from a list or array of elements. In an array, elements are stored in consecutive memory locations. Searching is done by comparing a particular element with the remaining elements until the exact match is found. If the element is found, the search process is said to be successful or else the search process is terminated unsuccessfully. The time complexity of searching technique depends on the number of comparisons made to find the exact match.

Binary Search

Binary search technique is implemented on sorted list of elements. It is faster than linear search. Hence, this method is efficient when the size of the list is larger.

The principle followed in binary search is "divide and conquer". The search is initiated by dividing the list into two sublists and computing the middle element. This element is then compared with the element to be searched. If the middle element is same as desired element (say, key) element, the search is terminated successfully, and the location (or index) of the middle element is returned. On the other hand, if the middle element doesn't match, then it is checked whether the middle element is greater than or less than the key element. If middle element is greater, then it is searched and performed on right sublist or else, the desired element is searched in the left sublist. This process is repeated, until the desired element is found. If the desired element is not found, then the search terminates unsuccessfully.

Algorithm

Input

L[ ] [i.e., Address of the list of elements]

n [i.e., the number of elements in the list]

key [i.e., the element to be searched]

Input: L[ ] (i.e., address of the list of elements),

In this search method, the list of elements is divided into two groups and the key element is searched in these groups. This process is carried out until the element is found in the list. If the key element is not found in the list then a message "Unsuccessful search, element not found" is displayed.

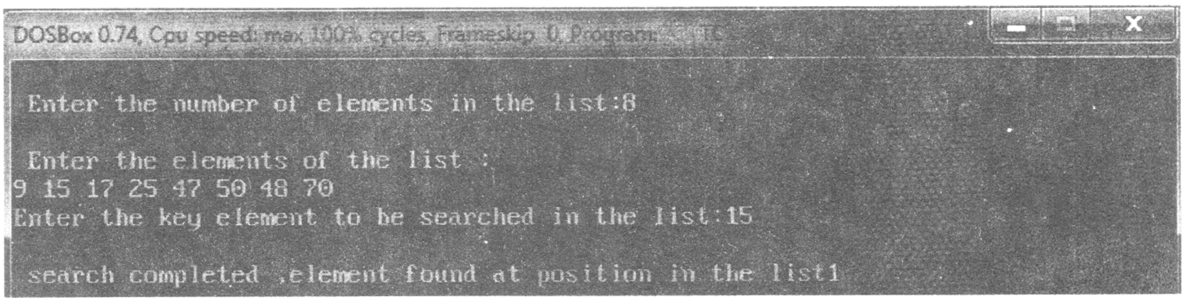
```
Step 1 :  Start
Step 2 :  Initialize low_val = 0 and high_val = n
          While (high_val > = low_val) do
          mid_val = (low_val + high_val)/2
          Goto step 3
          Otherwise, when condition fails
          cout<< "Unsuccessful search, element not found";
          Goto step 6
Step 3 :  Check, if (L[mid_val] = key) then
          cout<<"Element found at midval";
          Goto step 6
          else
Step4 :   Check, if (L[mid_val] > key) then
          high_val = mid_val – 1
          Goto step 2
Step5 :   Check, if (L[mid_val]] < key) then
          low_val = mid_val + 1
          Goto step 2
Step6 :   Stop.
```

Program

```
#include<iostream.h>
#include<conio.h>
int nr_bin_search(int [ ], int, int);
```

```
        int main()
        {
        clrscr();
        int key, i, n, index, 1[20];
        cout < <"\n Enter the number of elements in the list:";
        cin> >n;
        cout< <"\n Enter the elements of the list :\n";
        for(i=0; i <n; i++)
        cin> >![i];
        cout< <"Enter the key element to be searched in the list:";
        cin> >key;
        index=nr_bin_search(l, n, key);
        if(index=-l)
        cout< <"\n search completed ,element not found";
        else
        cout< <"\n search completed .element found at position in the list"< <index;
        getch();
        return 0;
}
        int nr_bin_search(int s[ ], int n, int e)
{
        int low val ,mid_val ,high_val,x;
        low_val=0;
        highval = n;
        while(high_val> =low_val)
{
        mid_val=(low_val + high_val)/2;
        if (s[mid_val]= =e)
        retum(midval);
        if(s[mid_val]<e)
        low_val=mid_val+1;
        else
        high val = mid val - 1;
        }
        return -1;
}
```

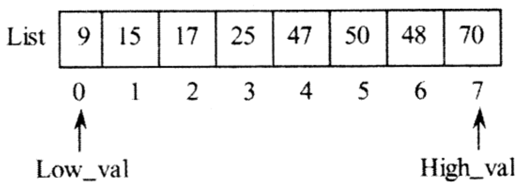
Output



```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

 Enter the number of elements in the list:8

 Enter the elements of the list :
9 15 17 25 47 50 48 70
Enter the key element to be searched in the list:15

 search completed .element found at position in the list1
```

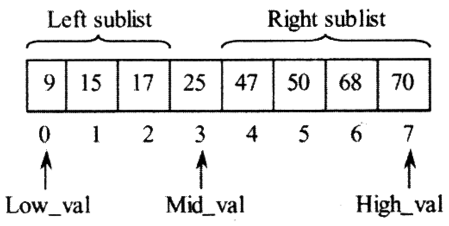
Example

Consider the following ordered list having 8 elements.

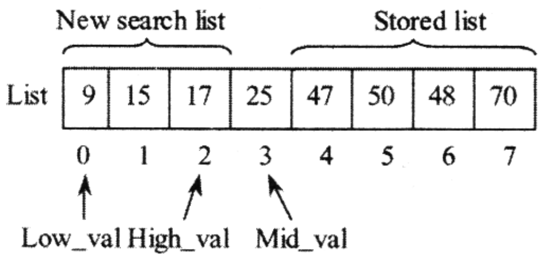


Assume that user wants to search element ‘15’. Initially, the middle (or centre) element is determined by using the formula:

$$\text{mid val} = \frac{\text{Low\_value} + \text{High\_value}}{2} = \frac{0+7}{2} = 3$$



Now, we compare key Ele i.e., 15, with the value at index ‘mid_ val’ i.e., 25. Since, $25 \neq 15$ and moreover 15 < 25, user look for ‘keyEle’ in left sublist. In other words, user restrict our search to the new list i.e., list [0] to list [2], This is done, by moving the high val pointer to one location before the mid val pointer.



In the next step, user calculates the middle element (i.e., mid val) of the new list as,

$$\text{Mid} - \text{val} = \frac{\text{Low\_value} + \text{High\_value}}{2} = \frac{0+2}{2} = 1$$

Therefore, the mid_val found is at index I, i.e., List[I] = 15.

Now, compare the 'keyEle' with the value at location mid val. Since, List[mid_val] is equal to 'KeyEle', then element is found. Hence, its location (i.e., pointed by mid val) is returned.

Time Complexities

A non-recursive binary search algorithm uses a while loop. The number of iterations takes place depends on the number of elements in the list. In a worst case, the search requires O(1og n) iterations where n is the number of elements in the list.

Q40. Give an algorithm and explain the concept of selection sort with an example program.

Ans :

Selection Sort

Selection sort is one of the simplest internal sorting technique. It involves selection of the smallest element from the given list and replacing it with the first element. This process is done on the unsorted portion of the list and will be repeated until all the elements in the list are sorted. Initially, the complete list will be unsorted. The selected smallest element is swapped with the first element. Now, the first element is said to be sorted and the remaining elements are unsorted. This process will be repeated on the unsorted portion until all the elements in the list are sorted.

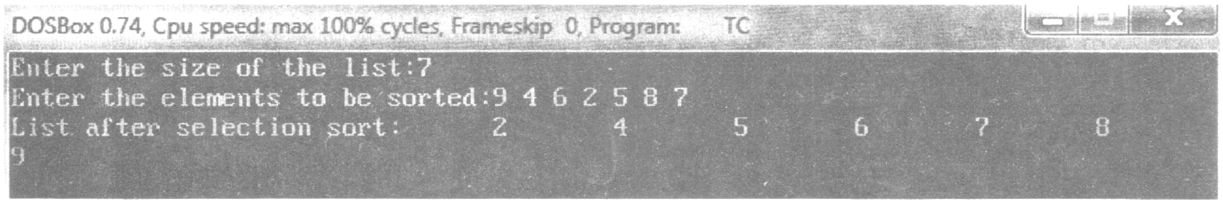
Algorithm for Selection Sort

```
Step 1: Read all the elements from the list.
Step 2 : Store each element in an array
Step 3 : Set the index value i=I
Step 4: Repeat step 5 to step 10 until all the elements in the list are sorted.
Step 5: Set first to i
Step 6: Set smallest to i+1
Step 7: Repeat step 8 and step 9 while smallest < size _ of _the_list.
Step 8 : If smallest < min, swap min and smallest.
Step 9: Increment smallest
Step 10: Increment i
Step 11: End
```

Program

```
        #include<iostream.h>
        #include<conio.h>
        int main()
{
        int i,min,n,next,temp,sortList[20];
        clrscr();
        cout<<"Enter the size of the list:";
        cin>>n;
        cout<<"Enter the elements to be sorted:";
        for(i=0;i<n;i++)
        cin>>sort List[i];
        for(i=0;i<n;i++)
        {
        for(next=i+1 ;next<n;next++)
        {
        min=i;
        if(sortList[next]<sortList[min])
        {
        temp=sortList[min];

        sortList[min]=sortListfnext];
        sortList[next]=temp;
        }
}
}
        cout<<"List after selection sort:";
        for(i=0;i<n;i++)
        cout<<"\t"<<sortList[i];
        getch(); return 0;
}
```

Output



```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program:    TC
Enter the size of the list:7
Enter the elements to be sorted:9 4 6 2 5 8 7
List after selection sort:      2       4       5       6       7       8
9
```

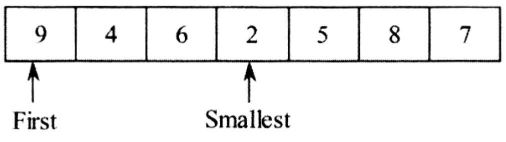
Example

Consider the following unsorted list.

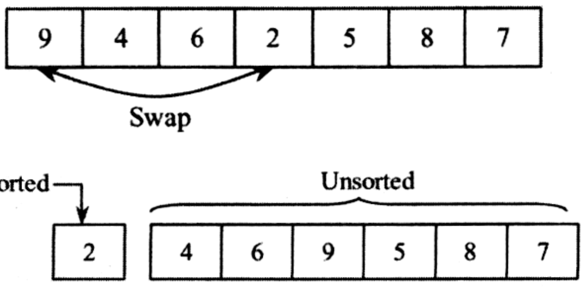
9	4	6	2	5	8	7

Iteration 1

The list is scanned for the smallest element.



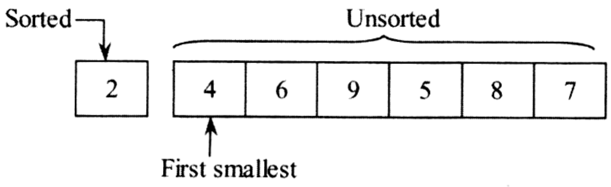
2 is found to be the smallest element and it is swapped with first element (i.e., 9) in the unsorted list



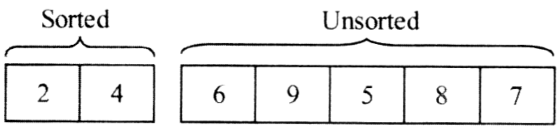
After swapping 2 becomes the sorted item.

Iteration 2

The unsorted portion of the list is scanned for the smallest element.

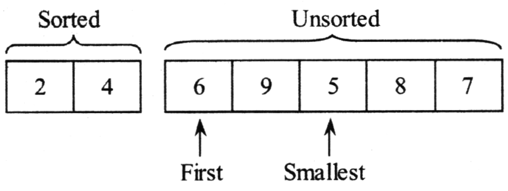


The smallest element found is 4. It has to be swapped with the first element in the unsorted portion of the list But, 4 is found to be first element so, there is no need of swapping. And, 4 is moved to the sorted portion of the list

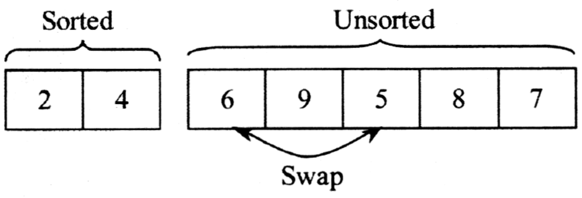


Iteration 3

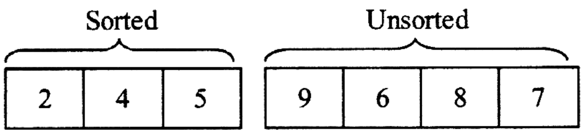
Scan the unsorted portion for the smallest item.



The smallest element found is 5. Swap it with the first element i.e., 6.

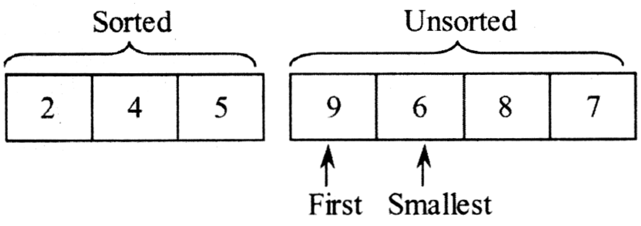


The lists after wapping is as follows,

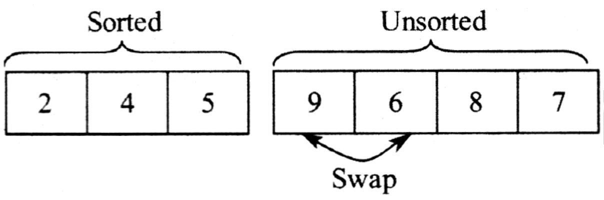


Iteration 4

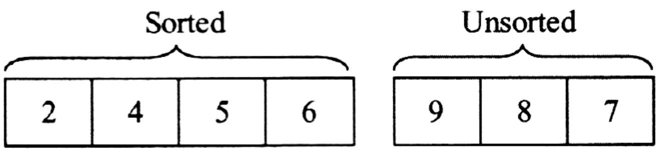
Scan the unsorted list for the smallest element.



The smallest element found is 6, swap it with the first element i.e., 9

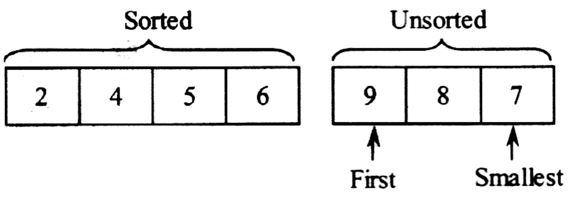


The list after swapping is as follows,

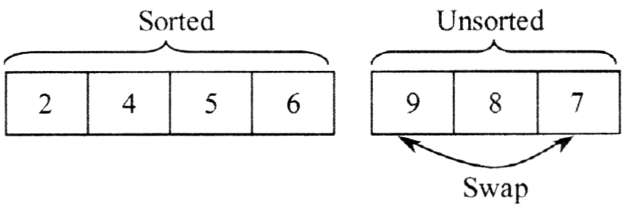


Iteration 5

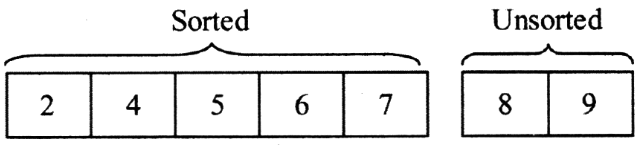
Scan the unsorted list for the smallest element.



Swap the smallest element 7 with the first element in the list 9.

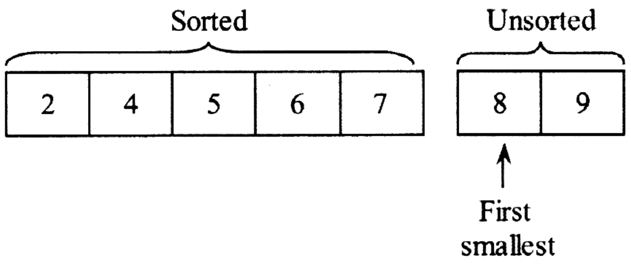


The list after swapping is as follows,

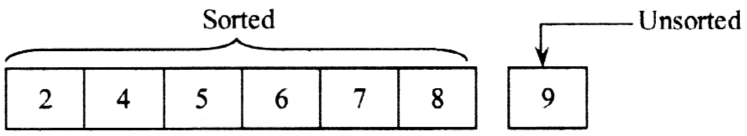


Iteration 6

Scan for the smallest element in the unsorted list.

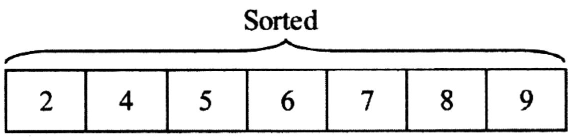


The smallest element found is 8. Swap it with the first element . The smallest element 8 is itself the first element. So, there is no need of swapping.



Iteration 7

Since there is only one element in the unsorted portion, it will automatically be placed in its proper placed in its proper place. So there is no need of swapping.



1.1.10 Strings

Q41. Define string? How to define it in C++?

Ans :

In C programming, the collection of characters is stored in the form of arrays, this is also supported in C++ programming. Hence it's called C-strings.

C-strings are arrays of type char terminated with null character, that is, \0 (ASCII value of null character is 0).

Syntaext

char str[] = "C++";

In the above code, str is a string and it holds 4 characters.

Although, "C++" has 3 character, the null character \0 is added to the end of the string automatically.

Alternative ways of defining a string

```
char str[4] = “C++”;
char str[] = {‘C’,’+’,’+’,’\0'};
char str[4] = {‘C’,’+’,’+’,’\0'};
```

Like arrays, it is not necessary to use all the space allocated for the string.

Example

```
char str[100] = “C++”;
```

Q42. Write a C++ program to display a string entered by user.

Ans :

```
#include<iostream>
usingnamespace std;
int main()
{
char str[100];
   cout <<“Enter a string: “;
   cin >> str;
   cout <<“You entered: “<< str << endl;

   cout <<“\nEnter another string: “;
   cin >> str;
   cout <<“You entered: “<<str<<endl;
return0;
}
```

Output

```
Enter a string: C++
You entered: C++
Enter another string: Programming is fun.
You entered: Programming
```

Q43. Write a C++ program to read and display an entire line entered by user.

Ans :

```
#include<iostream>
usingnamespace std;
int main()
{
char str[100];
   cout <<“Enter a string: “;
```

```
    cin.get(str,100);
    cout <<"You entered: "<< str << endl;
return0;
}
```

Output

```
        Enter a string: Programming is fun.
        You entered: Programming is fun.
```

Q44. What are the various string functions used in C++.

Ans :

C++ supports a wide range of functions that manipulate null-terminated strings:

S.No.	Function & Purpose
1	**strcpy(s1, s2);**Copies string s2 into string s1.
2	**strcat(s1, s2);**Concatenates string s2 onto the end of string s1.
3	**strlen(s1);**Returns the length of string s1.
4	**strcmp(s1, s2);**Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);**Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);**Returns a pointer to the first occurrence of string s2 in string s1.

Q45. Write a C++ program to perform string copy, concatenation and find stirng length using string class.

Ans :

```
#include <iostream>
#include <string>
using namespace std;
int main () {
   string str1 = "Hello";
   string str2 = "World";
   string str3;
   int  len ;
   // copy str1 into str3
   str3 = str1;
   cout << "str3 : "<< str3 << endl;
   // concatenates str1 and str2
   str3 = str1 + str2;
   cout << "str1 + str2 : "<< str3 << endl;
   // total lenghth of str3 after concatenation
   len = str3.size();
   cout << "str3.size() :  " << len << endl;
   return 0;
}
```

OUTPUT

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() :  10
```

Q46. Write a program to Find Frequency of Characters in a C-style String.

Ans :

```
#include<iostream>
usingnamespace std;
int main()
{
        char c[]="C++ programming is not easy.", check ='m';
        int count =0;
        for(int i =0; c[i]!='\0';++i)
{
        if(check == c[i])
        ++count;
}
        cout <<"Frequency of "<< check <<" = "<< count;
        return0;
}
```

Output

```
Number of m = 2.
```

Q47. Write a C++ program to find the number of vowels, consonants, digits and white-spaces.

Ans :

```
#include<iostream>
usingnamespace std;
int main()
{
string line;
int vowels,consonants, digits, spaces;
```

```
    vowels =consonants= digits = spaces =0;
    cout <<"Enter a line of string: ";
    getline(cin, line);
for(int i =0; i < line.length();++i)
{
if(line[i]=='a'|| line[i]=='e'|| line[i]=='i'||
           line[i]=='o'|| line[i]=='u'|| line[i]=='A'||
           line[i]=='E'|| line[i]=='I'|| line[i]=='O'||
           line[i]=='U')
{
++vowels;
}
elseif((line[i]>='a'&& line[i]<='z')||(line[i]>='A'&& line[i]<='Z'))
{
++consonants;
}
elseif(line[i]>='0'&& line[i]<='9')
{
++digits;
}
elseif(line[i]==' ')
{
++spaces;
}
}
    cout <<"Vowels: "<< vowels << endl;
    cout <<"Consonants: "<<consonants<< endl;
    cout <<"Digits: "<< digits << endl;
    cout <<"White spaces: "<< spaces << endl;
return0;
}
```

Output

```
Enter a line of string: I have 2 C++ programming books.
Vowels: 8
Consonants: 14
Digits: 1
White spaces: 5
```

1.1.11 Pointers

Q48. Define

i) Pointer

ii) Pointer value

Ans :

i) Pointer

A pointer is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it.

ii) Pointer Value

Whenever a variable is declared, system will allocate a location to that variable in the memory, to hold value. This location will have its own address number.

For example, we declare a variable of type integer with the name a by writing:

```
int a = 10;
```

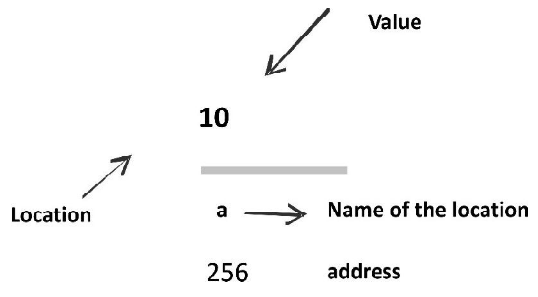
On seeing the "int" part of this statement the compiler reserves 2 bytes of memory from the addresses to hold the value of the integer.

The value 10 will be placed in that memory location reserved for a.

These memory locations assigned to the variables by the system are called pointer values. The address 256is assigned to the variable a is a pointer value.

Let us assume that system has allocated memory location 256 for a variable a, which is called a pointer value.

Value



Explanation

Consider the above example, where we have used to print the address of the variable using ampersand operator.

In order to print the variable we simply use name of variable while to print the address of the variable we use ampersand along with.

Q49. What is pointer variable ? how to declare and initialise pointer variable.

Ans :

Pointer Variable

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Value of pointer variable will be stored in another memory location.

Accessing a variable through Pointer

For accessing the variables through pointers,

The following steps to be performed to access pointers

➢ Declare a pointer variable

➢ Initializing of a variable to a pointer and

➢ Finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand.

Declaring a pointer variable

In C , every variable must be declared before they are used. A pointer variable can store only the address of the variable which has the same data-type as the pointer variable.

Syntax: data_type_name * variable name

Example : int *ptr;

Explanation

In the above example we have declare the pointer variable with the name of 'ptr' and its data-type in int, that means it can store the address of the variable which is of integer type.

This tells the compiler three things about the variable ptr.

1. The asterisk(*) tells that the variable ptr is a pointer variable.
2. ptr needs a memory location.
3. ptr points to a variable of type data type.

Different ways of declaring Pointer Variable

```
Ex:  Int   *p;
     Int   * p;
     Int   * p;
```

****Note :** * can appears anywhere between Pointer_name and Data Type

Example of declaring integer pointer

int a= 10;

int *ptr

Example of declaring character pointer

char ch='A';

char *cptr;

Example of declaring float pointer

float pi= 3.14;

float *fptr;

Q50. Explain how to initialise pointers.

Ans :

Initializing Pointers

Once a pointer variable has been declared ,it must be assigned some value. The process of assigning the address of a variable to a pointer variable is known as initialization.

The initialization of the pointer variable is simple like other variable but in the pointer variable we assign the address instead of value.

Initialization of pointer can be done using 4 steps

1. Declare a Pointer Variable and Note down the Data Type.
2. Declare another Variable with Same Data Type as that of Pointer Variable.
3. Initialize Ordinary Variable and assign some value to it.
4. Now Initialize pointer by assigning the address of ordinary variable to pointer variable.

Example

```
int   *ptr,
int a = 10 ;
ptr=&a;
```

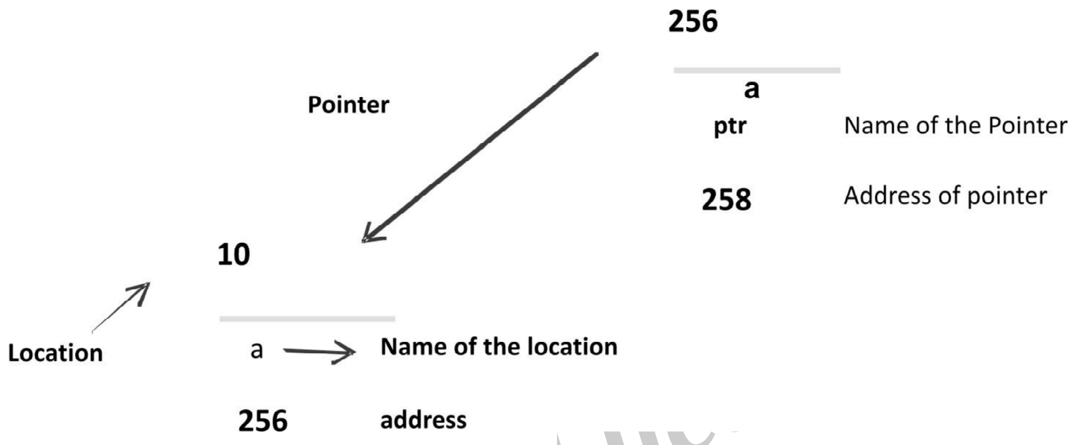
****Note:** Pointers are always initialized before using it in the program

Explanation

Here in the above example we have a pointer variable ptr and another is a simple integer variable a, and we have assign the pointer variable with the address of the 'a'. That means the pointer variable 'ptr' is now has the address of the variable 'a'.

**** Note:** using the '*' asterisk sign before the pointer variable means the pointer variable is now pointing to the value at the location instead of the pointing to location.

Let us assume that syste m has allocated memory location 256 for a variable a, which is called a pointer value. Here the variable ptr is declared to hold the value of address of a variable 'a' which is allocated by the compiler.



```
// Demonstrating Declaring and Initialization of pointer
#include<iostream.h>
int main()
{
int a = 10;
int *ptr;
ptr = &a;
cout<< "\nValue of ptr : %u",ptr;
return(0);    }
```

Q51. What are Reference operator and Deference operators in C++.

Ans :

The Address (&) Operator

The address of the variable cannot be accessed directly.

1. Pointer address operator is denoted by '&' symbol
2. When we use ampersand symbol as a prefix to a variable name '&', it gives the address of that variable.

The format specifier of address is %u(unsigned integer), because the addresses are always positive values.

Example: &a - It gives an address on variable a

```
// Demonstrating the address operator
#include<iostream.h>
void main()
{
int n = 10;
cout << "\nValue of a is : %d",a;
cout << "\nValue of &a is : %u",&a;
}
```

Output

```
Value of a is : 10
Value of &a is : 256
```

The * Operator

1. In order to create pointer to a variable we use "*" operator .
2. '*' is called as 'Value at address' Operator
3. 'Value at address' Operator gives 'Value stored at Particular address.
4. 'Value at address' is also called as 'Indirection Operator' or 'Dereferencing Operator'

Consider the previous example, We can access the value 10 by either using the variable name a or the address 256. The variable that holds memory address are called pointer variables.

1.2 Functions

1.2.1 Introduction

Q52. Define function. Write how a function can be declared and defined.

Ans :

Function

A function is a program segment that performs specific and well defined tasks. Many of the functions are reusable i.e., they can be defined once and executed for desired number of times by calling it from any part of the program.

Function Declaration

A function declaration declares a function that consists of a function type (or return type), function name parameter (or argument) list and a terminating semicolon. Function declaration is also called as function prototype. A function must be declared before it is invoked.

Syntax

Function type functionname (parameter_list);

Example

```
float sum(float x, float y);
```

This example is a function prototype for the function sum() that takes two float arguments and returns a float value as result.

Function Definition

A function definition is the complete description of a function., a function definition tells what a function does and how it performs. A function definition contains a function body (a block of statements) in addition to function name, arguments list and return type.

Syntax

```
retumtype fimction_name(parameter_list)
{
    local variable declarations;
    //statements;
    return (expression);
}
```

Example

```
float sum(float x, floaty)
{
    float Z;
    z = x + y;
    return (z);
}
```

1.2.2 Prototype

Q53. Define function prototype.

Ans :

Function prototype(declaration)

If an user-defined function is defined after main() function, compiler will show error. It is because compiler is unaware of user-defined function, types of argument passed to function and return type.

In C++, function prototype is a declaration of function without function body to give compiler information about user-defined function. Function prototype in above example:

```
int add(int, int);
```

You can see that, there is no body of function in prototype. Also there are only return type of arguments but no arguments. You can also declare function prototype as below but it's not necessary to write arguments.

```
int add(int a, int b);
```

*****Note:** It is not necessary to define prototype if user-defined function exists before **main()**function.

Function Definition

The function itself is referred as function definition. Function definition in the above program:

```
/* Function definition */
 int add(int a,int b) { // Function declarator
 int add;
 add = a+b;
 return add;          // Return statement
}
```

When the function is called, control is transferred to first statement of function body. Then, other statements in function body are executed sequentially. When all codes inside function definition is executed, control of program moves to the calling program.

Q54. What is Function Call?

Ans :

To execute the codes of function body, the user-defined function needs to be invoked(called). In the above program, add(num1, num2); inside main() function calls the user-defined function. In the above program, user-defined function returns an integer which is stored in variable add.

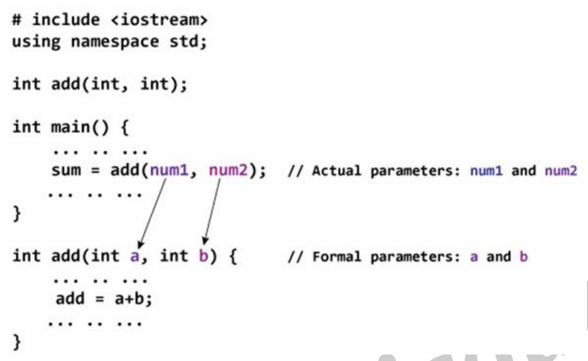
1.2.3 Passing Data by Value

Q55. Elucidate the pass arguments to function.

Ans :

In programming, argument(parameter) refers to data this is passed to function(function definition) while calling function.

In above example, two variables, num1 and num2 are passed to function during function call. These arguments are known as actual arguments. The value of num1 and num2 are initialized to variables a and b respectively. These arguments a and b are called formal arguments. This is demonstrated in figure below:



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

int add(int, int);

int main() {
    ... .. ...
    sum = add(num1, num2);  // Actual parameters: num1 and num2
   ... .. ...
}

int add(int a, int b) {     // Formal parameters: a and b
    ... .. ...
     add = a+b;
   ... .. ...
}
```

Points to Remember

- ➢ The numbers of actual arguments and formals argument should be same. (Exception: Function Overloading)
- ➢ The type of first actual argument should match the type of first formal argument. Similarly, type of second actual argument should match the type of second formal argument and so on.
- ➢ You may call function without passing any argument. The number(s) of argument passed to a function depends on how programmer want to solve the problem.
- ➢ In above program, both arguments are of int type. But it's not necessary to have both arguments of same type.

Q56. Explain how to pass parameters by calling a Function

(OR)

Explain about the concepts of call by value and call by reference

Ans :

Functions are called by their names. If the function is without argument, it can be called directly using its name. But for functions with arguments, we have two ways to call them,

1. Call by Value
2. Call by Reference

1. Call by Value

In this calling technique we pass the values of arguments which are stored or copied into the formal parameters of functions. Hence, the original values are unchanged only the parameters inside function changes.

```
void calc(int x);
int main()
{
 int x = 10;
 calc(x);
 printf("%d", x);
}
void calc(int x)
{
 x = x + 10 ;
}
```

Output : 10

In this case the actual variable x is not changed, because we pass argument by value, hence a copy of x is passed, which is changed, and that copied value is destroyed as the function ends(goes out of scope). So the variable x inside main() still has a value 10.

But we can change this program to modify the original x, by making the function calc() return a value, and storing that value in x.

```
int calc(int x);
int main()
{
 int x = 10;
 x = calc(x);
 printf("%d", x);
}
int calc(int x)
{
 x = x + 10 ;
 return x;
}
```

Output : 20

2. Call by Reference

In this we pass the address of the variable as arguments. In this case the formal parameter can be taken as a reference or a pointer, in both the case they will change the values of the original variable.

```
void calc(int *p);
int main()
{
 int x = 10;
 calc(&x);     // passing address of x as argument
 printf("%d", x);
}
void calc(int *p)
{
 *p = *p + 10;
}
```

Output : 20

1.2.4 Reference Variables

Q57. What is Reference Variable ? What is its major use.

Ans :

C ++ introduces a new kind of varibale known as the reference variable. A reference variable provides an alias (alternative name) for a previously defined variable For example, if we make the variable sum a reference to the variable total. then sum and total can be used interchangeably to represent that variable . A reference variable is creatrd as follows :

$\boxed{\text{data – type \& reference - name = variable - name}}$

Example :

```
float total = 100 ;
float & sum = total ;
```

total is a float type variable that has already been declared; sum is the alternative name declared to represent the variable total. Both the variables refer to the same data object in the memory. Now, the both print the value 100. The statement

```
cout << total;
```

and

```
cout << sum;
```

both print the value 100. The statement

```
total = total + 10;
```

will change the value of both total and sum to 110. Likewise, the assignment

Sum = 0;

will change the value of both the variables to zero.

A reference variable must be initialized at the time of declaration. This establishes the correspon-dence between the reference and the data object which it names. It is important to note that the initialization of a reference variable is completely different from assignment to it.

C++ assigns additional meaning to the symbol &. Here, & is not an address operator. The notation float & means reference to float. Other examples are:

```
int n (10) ;
int & x = n [10];       / / x is alias for n[10]
char  & a = '\n';       / /initialize reference to a literal
```

The variable x is an alternative to the array element n[10]. The variable a is initialized to the newline constant. This creates a reference to the otherwise unknown location where the newline constant \n is stored.

The following references are also allowed:

```
i)   int x ;
     int * p = 6 x ;
ii)  int * m = *p;
```

The first set of declarations causes m to refer to x which is pointed to by the pointer p and the statement in (ii) creates an int object with value 50 and name n.

A major application of reference variables is in passing arguments to functions. Consider the following :

```
void f (int & x)        / / uses reference
{
     x = x = 10;        / / x is incremented ;  so also m
}
int main (  )
{
     int m = 10;
     f(m)  ;            / / function call
Object Oriented Programming with C ++
........
........
}
```

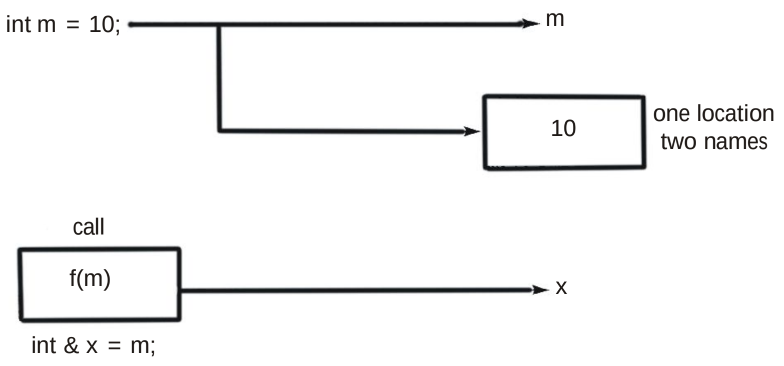
When the function call f(m) is executed, following initialization occurs :

int & x = m ;

Thus, x becomes an alias of m after executing the statement

f(m);

Such function calls are known as call by reference. This implementation is illustrated in figure since the variables x and m are aliases, when the function increments x, m is also incremented. The value of m becomes 20 after the function is executed. In traditional C, we accomplish this operation using pointers and dereferencing techniques.



Call by reference mechanism

The call by reference mechanism is useful in object-oriented programming because it permits the manipulation of objects by reference, and eliminates the copying of object parameters back and forth. It is also important to note that references can be created not only for built-in data types but also for user-defined data types such as structures and classes. References work wonderfully well with these user-defined data types.

1.2.5 Using Reference Variables as Parameters

Q58. Write a program to Pass parameters by references in C++.

Ans :

```
#include <iostream>
using namespace std;
    // function declaration
void swap(int& x, int& y);
int main () {
    // local variable declaration:
int a = 100;
int b = 200;
cout << "Before swap, value of a :"
    << a << endl;
cout << "Before swap, value of b :" << b << endl;
/* calling a function to swap the values.*/
swap(a, b);
cout << "After swap, value of a :" << a << endl;
cout << "After swap, value of b :" << b << endl;
return 0;
}
    // function definition to swap the values.
void swap(int& x, int& y) {
```

```
int temp;
temp = x; /* save the value at address x */
x = y;    /* put y into x */
y = temp; /* put x into y */
return;
}
```

When the above code is compiled and executed, it produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

1.2.6 Inline Functions

Q59. What are called as Inline Functions?

Ans :

Inline functions are actual functions, which are copied everywhere during compilation, like preprocessor macro, so the overhead of function calling is reduced. All the functions defined inside class definition are by default inline, but you can also make any non-class function inline by using keyword **inline** with them.

For an inline function, declaration and definition must be done together. For example,

```
inline void fun(int a)
{
    return a++;
}
```

Some Important points about Inline Functions

1. We must keep inline functions small, small inline functions have better efficiency.
2. Inline functions do increase efficiency, but we should not make all the functions inline. Because if we make large functions inline, it may lead to code bloat, and might affect the speed too.
3. Hence, it is adviced to define large functions outside the class definition using scope resolution ::operator, because if we define such functions inside class definition, then they become inline automatically.
4. Inline functions are kept in the Symbol Table by the compiler, and all the call for such functions is taken care at compile time.

Limitations of Inline Functions

1. Large Inline functions cause Cache misses and affect performance negatively.
2. Compilation overhead of copying the function body everywhere in the code on compilation, which is negligible for small programs, but it makes a difference in large code bases.
3. Also, if we require address of the function in program, compiler cannot perform inlining on such functions. Because for providing address to a function, compiler will have to allocate storage to it. But inline functions doesn't get storage, they are kept in Symbol table.

Q60. Write a program to find the multiplication values and the cubic values using inline function.

Ans :

```
#include<iostream.h>
#include<conio.h>
class line
{
  public:
          inline float mul(float x,float y)
          {
                    return(x*y);
          }
          inline float cube(float x)
          {
                    return(x*x*x);
          }
};
void main()
{
          line obj;
          float val1,val2;
          clrscr();
          cout<<"Enter two values:";
          cin>>val1>>val2;
          cout<<"\nMultiplication value
is:"<<obj.mul(val1,val2);
          cout<<"\n\nCube value is :"<<obj.cube(val1)
            <<"\t"<<obj.cube(val2);
          getch();
}
```

Output:

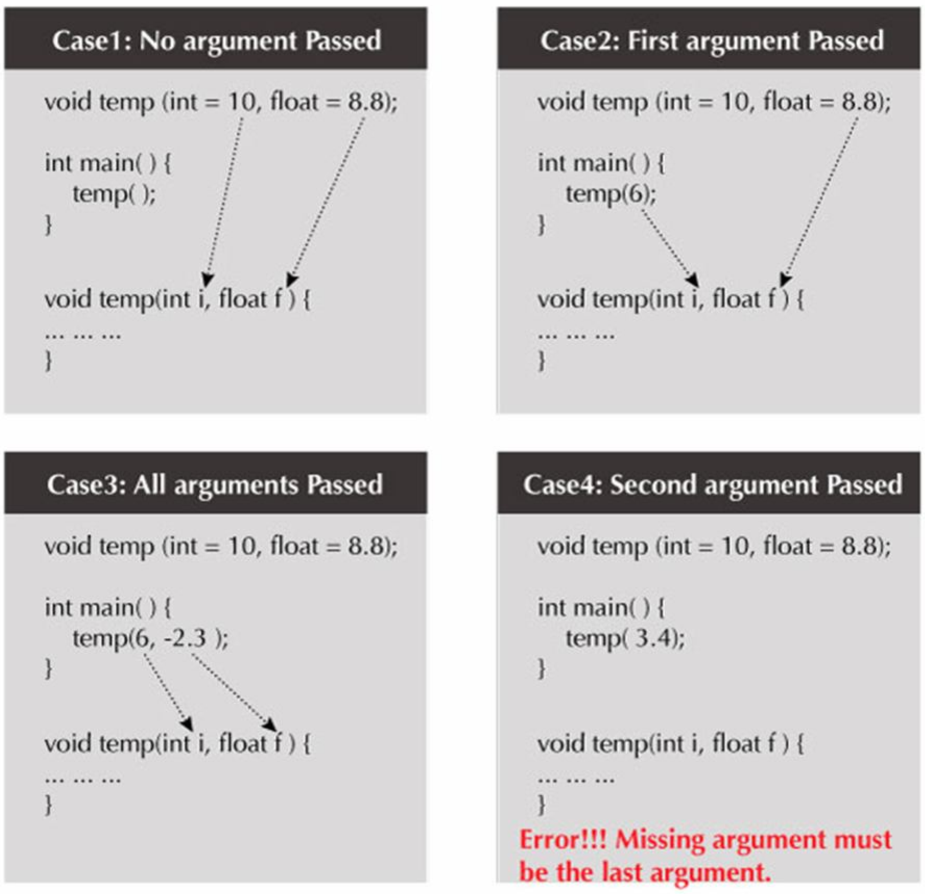
```
Enter two values: 5  7
Multiplication Value is: 35
Cube Value is: 25 and 343
```

1.2.7 Default Arguments

Q61. What are Default Argumentsin C++ . explain them.

Ans :

In C++ programming, you can provide default values for function parameters. The idea behind default argument is very simple. If a function is called by passing argument/s, those arguments are used by the function. But if all argument/s are not passed while invoking a function then, the default value passed to arguments are used. Default value/s are passed to argument/s in function prototype. Working of default argument is demonstrated in the figure below:



Case1: No argument Passed

```
void temp (int = 10, float = 8.8);

int main( ) {
    temp( );
}

void temp(int i, float f ) {
... ... ...
}
```

Case2: First argument Passed

```
void temp (int = 10, float = 8.8);

int main( ) {
    temp(6);
}

void temp(int i, float f ) {
... ... ...
}
```

Case3: All arguments Passed

```
void temp (int = 10, float = 8.8);

int main( ) {
    temp(6, -2.3 );
}

void temp(int i, float f ) {
... ... ...
}
```

Case4: Second argument Passed

```
void temp (int = 10, float = 8.8);

int main( ) {
    temp( 3.4);
}

void temp(int i, float f ) {
... ... ...
}
```

Error!!! Missing argument must be the last argument.

Fig.: Working of Default Argument in C++

/*C++ Program to demonstrate working of default argument */

```
#include<iostream>
usingnamespace std;
void display(char='*',int=1);
int main(){
   cout<<"No argument passed:\n";
   display();
   cout<<"\n\nFirst argument passed:\n";
   display('#');
       cout<<"\n\nBoth argument passed:\n";
   display('$',5);
return0   }  void display(char c,int n)
{
for(int i =1; i <=n;++i)
{
       cout<<c;
}     cout<<endl;
}
```

Output

```
No argument passed:
*
First argument passed:
#
Both argument passed:
$$$$$
```

In the above program, At first, display() function is called without passing any arguments. In this case, default() function used both default arguments. Then, the function is called using only first argument. In this case, function does not use first default value passed. Function uses the actual parameter passed as first argument and takes default value(second value in function prototype) as it's second argument. When display() is invoked passing both arguments, default arguments are not used.

***Note: The missing argument must be the last argument of the list, that is, if you are passing only one argument in the above function, it should be the first argument.

1.2.8 Overloading Functions

Q62. Explain briefly about overloading of a function.

Ans :

Overloading refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in OOP.

Using the concept of function overloading; we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type. For example, an overloaded add() function handles different types of data as shown below:

```
/ /Declarations
int add(int a, int b) ;                / / prototype 1
int add(int a, int b, int c) ;         / / prototype 2
double add(double x, double y);        / / prototype 3
double add(int p, double q) ;          / / prototype 4
double add(double p, int q);           / / prototype 5
/ / Function calls
 cout < < add(5, 10);                  / / uses prototype 1
 cout < <  add(15, 10.0);              / / uses prototype 4
cout < < add (12.5, 7.5);              / / uses prototype 3
cout < < add (5, 10, 15);              / / uses prototype 2
cout < < and  (0.75, 5);               / / uses prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The ftinction selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.

2. If an exact match is not found, the compiler uses the integral promotions to the actual argument,

   such as,

   char 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.

   If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

   long square (long n)

   double square (double x)

   A function call such as

   square (10)

   will cause an error because int argument can be converted to either long or double, thereby creating an ambiguous situation as to which version of square() should be used.

4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

Program:

```
/ / Function volume ( ) is overloaded three times
# include < isostream>
using namespace std ;
/ / Declarations (prototypes)
int volume (int);
double volume (double, int);
long volume (long, int, int);
int main ( )
{
    cout< < " Caling the volume ( ) volume ( ) function for computing the volume of a cube -
        " < <volume (10)< <"\n" ;
    cout < <"Calling the volume ( ) function for computing the volume of a cylinder - "< <volume
        (2.5,8)< <"\n";
    cout < <" calling the volume ( ) function for computing the volume of a rectangular box -
    "< < volume (100L, 75, 15);
return 0;
/ / Funcion definitions
int volume (int s) // cube
{
    return (s * s * s);
}
double volume (double r, int h) // cylinder
    return (3. 14519 * r * r * h);
}
long volume (long 1, int b, int h) / / rectangular box
{
    return (1* b* h);
}
```

Output

```
Calling the volume ( ) function for computing the volume of a cube - 1000
Calling the volume ( ) function for computing the volume of a cylinder - 157.26
Calling the volume ( ) function for computing the volume of a rectangular box - 112500
```

Q63. Write a program to demonstrate Function Overloading.

Ans :

```
#include<iostream>
usingnamespace std;
void display(int);
void display(float);
void display(int,float);
int main(){
int a =5;
float b =5.5;
   display(a);
   display(b);
   display(a, b);
return0;
}
void display(intvar){
       cout <<"Integer number: "<<var<< endl;
}
void display(floatvar){
       cout <<"Float number: "<<var<< endl;
}
void display(int var1,float var2){
       cout <<"Integer number: "<< var1;
       cout <<" and float number:"<< var2;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

Here, the display() function is called three times with different type or number of arguments.

The return type of all these functions are same but it's not necessary.

Q64. Write a program to compute absolute value using function overloading.

Ans :

```
// Program to compute absolute value
// Works both for integer and float
#include<iostream>
usingnamespace std;
int absolute(int);
float absolute(float);
int main(){
int a =-5;
float b =5.5;
   cout <<"Absolute value of "<< a <<" = "<<
            absolute(a)<< endl;
   cout <<"Absolute value of "<< b <<" = "<<
              absolute(b);
return0;
}
int absolute(intvar){
if(var<0)
var=-var;
returnvar;
}
float absolute(floatvar){
if(var<0.0)
var=-var;
returnvar;
}
```

Output

```
Absolute value of -5 = 5
Absolute value of 5.5 = 5.5
```

1.2.9 Passing Arrays to Functions

Q65. What are the different methods to pass arrays to functions.

Ans :

If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.

Method-1

Formal parameters as a pointer as follows:

```
void myFunction(int*param){
.
.
.
}
```

Method-2

Formal parameters as a sized array as follows:

```
void myFunction(int param[10]){
.
.
.
}
```

Method-3

Formal parameters as an unsized array as follows:

```
void myFunction(int param[]){
.
.
.
}
```

Now, consider the following function, which will take an array as an argument along with another argument and based on the passed arguments, it will return average of the numbers passed through the array as follows:

```
double getAverage(int arr[],int size){
int    i, sum =0;
double avg;
for(i =0; i < size;++i){
      sum += arr[i];
}
   avg =double(sum)/ size;
return avg;
}
```

Now, let us call the above function as follows:

```
#include<iostream>
usingnamespace std;
// function declaration:
double getAverage(int arr[],int size);
int main (){
// an int array with 5 elements.
int balance[5]={1000,2,3,17,50};
double avg;
// pass pointer to the array as an argument.
         avg = getAverage( balance,5);
// output the returned value
         cout <<"Average value is: "<< avg <<
                  endl;
         return0;
}
```

When the above code is compiled together and executed, it produces the following result:

```
Average value is: 214.4
```

Q66. Write a C++ Program to display marks of 5 students by passing one-dimensional array to a function.

Ans :

```
#include <iostream>
using namespace std;
void display(int marks[5]);
int main()
{
    int marks[5] = {88, 76, 90, 61, 69};
    display(marks);
    return 0;
}
void display(int m[5])
{
    cout << "Displaying marks: "<< endl;
    for (int i = 0; i < 5; ++i)
    {
      cout << "Student "<< i + 1 <<": "<< m[i]
<< endl;
    }
}
```

Output

```
Displaying marks:
Student 1: 88
Student 2: 76
Student 3: 90
Student 4: 61
Student 5: 69
```

Short Question & Answers

1. What is C++ ?

Ans :

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct feature of Simula67. Since the class was a major addition to the original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented (incremented) version of C.

2. What is variable?

Ans :

Variable is a location in the computer memory which can store data and is given a symbolic name for easy reference. The variables can be used to hold different values at different times during the execution of a program.

Basic types of Variables

Each variable while declaration must be given a datatype, on which the memory assigned to the variable depends. Following are the basic types of variables,

➢ bool For variable to store boolean values (True or False)

➢ char For variables to store character types.

➢ int for variable with integral values

➢ float and double are also types for variables with large and floating point values.

Declaration and Initialization

Variable must be declared before they are used. Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

3. Various operators in C++.

Ans :

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators.

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Miscellaneous Operators.

4. Explain while loop with syntax and example.

Ans :

while loop can be address as an **entry control** loop. It is completed in 3 steps. In **while loop** First check the condition if condition is true then control goes inside the loop body other wise goes outside the body. **while loop** will be repeats in clock wise direction.

➢ Variable initialization.(e.g int x=0;)

➢ condition(e.g while(x<=10))

➢ Variable increment or decrement (x++ or x— or x=x+2)

Syntax

```
variable initialization ;
while (condition)
{
 statements ;
 variable increment or decrement ;
}
```

***Note: If while loop condition never false then loop become infinite loop.

Example : print the natural numbers below 5

```
#include<iostream.h>
#include<conio.h>
void main()
{
int i;
clrscr();
i=1;
while(i<5)
{
cout<<endl<<i;
i++;
}
getch();
}
```

OUTPUT

```
1
2
3
4
```

5. Explain for loop with syntax and example.

Ans :

for loop is used to execute a set of statement repeatedly until a particular condition is satisfied.

For loop contains 3 parts.

- Initialization
- Condition
- Iteration

Syntax:

```
for(initialization; condition ; increment/decrement)
{
  statement-block;
}
```

- When we are working with for loop always execution process will start from initialization block.
- After initialization block control will pass to condition block, if condition is evaluated as true then control will pass to statement block.
- After execution of the statement block control will pass to iteration block, from iteration it will pass back to the condition.
- Always repetitions will happen beginning condition, statement block and iteration only.
- Initialization block will be executed only once when we are entering into the loop first time.
- When we are working with for loop everything is optional but mandatory to place 2 semicolons (; ;)

Example

```
while()        // Error
for( ; ; )     // valid
```

Example : print the natural numbers below 5 using for loop

```
#include<iostream.h>
#include<conio.h>
void main()
{
int i;
clrscr();
for(i=1;i<5;i++)
{
cout<<endl<<i;
}
getch();
}
```

OUTPUT

```
1
2
3
4
```

6. Overloading of a function.

Ans :

Overloading refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in OOP.

Using the concept of function overloading; we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type. For example, an overloaded add() function handles different types of data as shown below:

```
//Declarations
int add(int a, int b) ;                  // prototype 1
int add(int a, int b, int c) ;           // prototype 2
double add(double x, double y);          // prototype 3
double add(int p, double q) ;            // prototype 4
double add(double p, int q);             // prototype 5
// Function calls
 cout << add(5, 10);                     // uses prototype 1
 cout << add(15, 10.0);                  // uses prototype 4
cout << add (12.5, 7.5);                 // uses prototype 3
cout << add (5, 10, 15);                 // uses prototype 2
cout << and  (0.75, 5);                  // uses prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution.

7. What are called as Inline Functions?

Ans :

Inline functions are actual functions, which are copied everywhere during compilation, like preprocessor macro, so the overhead of function calling is reduced. All the functions defined inside class definition are by default inline, but you can also make any non-class function inline by using keyword inline with them.

For an inline function, declaration and definition must be done together. For example,

```
inline void fun(int a)
{
  return a++;
}
```

Some Important points about Inline Functions

1. We must keep inline functions small, small inline functions have better efficiency.
2. Inline functions do increase efficiency, but we should not make all the functions inline. Because if we make large functions inline, it may lead to code bloat, and might affect the speed too.

3. Hence, it is adviced to define large functions outside the class definition using scope resolution ::operator, because if we define such functions inside class definition, then they become inline automatically.

4. Inline functions are kept in the Symbol Table by the compiler, and all the call for such functions is taken care at compile time.

8. Write the simple Program of C++.

Ans :

```
# include < iostream > // include header file
using namespace std;
int main ( )
{
cout << "C ++ is better than c. \n"; / / C++ statement
return 0;
}     // End of example
```

9. What is reference variable in C++.

Ans :

Reference variable

Reference variable is a new feature added to C++. A reference variable basically assigns an alternative name to an existing variable.

The syntax for creating reference variable is :

Data_type & reference_name = variable_name

For ex: int n=10;

Int &num = nl

Then both n and num are the integer type variables and will store the same data.

10. Explain briefly about Control Structures.

Ans :

Control flow or flow of control is the order in which instructions, statements and function calls being executed or evaluated when a program is running. The control flow statements are also called as Flow Control Statements. In C++, statements inside your code are generally executed sequentially from top to bottom, in the order that they appear. It is not always the case your program statements to be executed straightforward one after another sequentially, you may require to execute or skip certain set of instructions based on condition, jump to another statements, or execute a set of statements repeatedly. In C++, control flow statements are used to alter, redirect, or to control the flow of program execution based on the application logic.

Multiple Choice Questions

1. Which datatype is used to represent the absence of parameters? [c]
   (a) int (b) short
   (c) void (d) float
2. Which type is best suited to represent the logical values? [b]
   (a) integer (b) boolean
   (c) character (d) all of the mentioned
3. What is this operator called ?: ? [a]
   (a) conditional (b) relational
   (c) casting operator (d) none of the mentioned
4. Which operator has the lowest precedence [d]
   (a) + (b) –
   (c) * (d) %
5. When the ________ statement is encountered then the rest of statements in the loop are skipped [c]
   (a) Break (b) Continue
   (c) Default (d) Case
6. Arguments of a functions are separated with [a]
   (a) comma (,) (b) semicolon
   (c) colon (d) None of these
7. The order in which actual arguments are evaluated in function call [c]
   (a) is from the left (b) is from the right
   (c) is compiler dependent (d) None of above
8. An array elements are always stored in ________ memory locations. [a]
   (a) Sequential (b) Random
   (c) Sequential and Random (d) None of the above
9. The declaration char (*a) [10]; means [b]
   (a) a is one dimensional array of size 10, of pointers to characters
   (b) a is a pointer to a 10 elements character array.
   (c) The same as char *a[10]
   (d) a is an array with 10 elements.
10. Which of the following gives the memory address of integer variable a? [c]
    (a) *a; (b) a;
    (c) &a; (d) address(a);

Fill in the blanks

1. ________ operator returns the number of bytes occupied by the operand.
2. ________ statements are used to repeat the execution of list of statements
3. Do-while is an ________ loop.
4. Dangling else problem occurs when ________
5. ________ is the index of the last element of an array with 9 elements ?
6. A function that calls itself for its processing is known as ________
7. We declare a function with ________ if it does not have any return type
8. Variables inside parenthesis of functions declarations have ________ level access.
9. ________ operator used for dereferencing or indirection
10. Null pointer is ________

ANSWERS

1. sizeof
2. Iterative
3. Exit control
4. there is no matching else
5. 8
6. Recursive Function
7. void
8. Local
9. " * "
10. A pointer which does not point anywhere

UNIT II

Object Oriented Programming: Procedural Programming verses Object-Oriented Programming, Terminology, Benefits, OOP Languages, and OOP Applications.

Classes: Introduction, Defining an Instance of a Class, Why Have Private Members? Separating Class Specification from Implementation, Inline Member Functions, Constructors, Passing Arguments to Constructors, Destructors, Overloading Constructors, Private Member Functions, Arrays of Objects, Instance and Static Members, Friends of Classes, Member-wise Assignment, Copy Constructors, Operator Overloading.

2.1 Object Oriented Programming

Q1. What do you understand by Object Oriented Programming?

Ans :

Object Oriented Programming

With the increase in size and complexity of programs, there was a need for a new programming paradigm that could help to develop maintainable programs. Consequently, OOP was developed. It treats data as a critical element in the program development and restricts its flow freely around the system. We have seen that monolithic, procedural, and structured programming paradigms are task-based as they focus on the actions the software should accomplish.

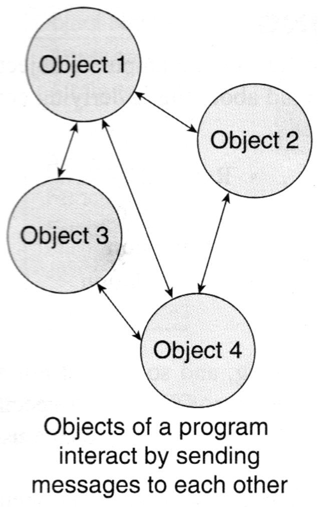

Object 1
Object 2
Object 3
Object 4
Objects of a program interact by sending messages to each other

Fig. : Object oriented paradigm

However, the object oriented paradigm is task-based and data-based. In this paradigm, all the relevant data and tasks are grouped together in entities known as objects.

For example, consider a list of numbers stored in an array. The procedural or structured programming paradigm considers this list as merely a collection of data. Any program that accesses this list must have some procedures or functions to process this list. For example, to find the largest number or to sort the numbers in the list, we needed specific procedures or functions to do the task. Therefore, the list was a passive entity was it is maintained by a controlling program rather than having the responsibility of maintaining itself.

However, in the object oriented paradigm, the list and the associated operations are treated as one entity known as an object. In this approach, the list is considered an object consisting of the list, along with a collection of routines for manipulating the list. In the list object, there may be routines for adding a number to the list, deleting a number from the list, sorting the list, etc.

The striking difference between OOP and traditional approaches is that the program accessing this list need not contain procedures for performing tasks; rather, it uses the routines provided in the object. In other words, instead of sorting the list as in the procedural paradigm, the program asks the list to sort itself.

Therefore, we can conclude that the object oriented paradigm is task-based (as it considers operations) as well as databased (as these operations are grouped with the relevant data).

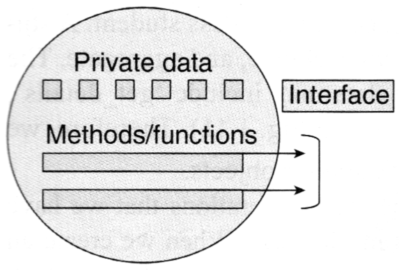


Fig. : Object

Above figure represents a generic object in the object oriented paradigm. Every object contains some data and the operations, methods, or functions that operate on that data. While some objects may contain only basic data types such as characters, integers, floating types, the other object, the other objects on the other hand may incorporate complex data types such as trees or graphs.

Programs that need the object will access the object's methods through a specific interface. The interface specifies how to send a message to the object, that is, a request for a certain operation to be performed.

For example, the interface for the list object may require that any message for adding a new number to the list should include the number to be added. Similarly, the interface might also require that any message for sorting specify whether the sort should be ascending or descending. Hence, an interface specifies how messages can be sent to the object.

Note : OOP is used for simulating real world problems on computers because real world is made of objects.

The striking features of OOP include the following:

- The programs are data centred.
- Programs are divided in terms of objects and not procedures.
- Functions that operate on data are tied together with the data.
- Data is hidden and not accessible by external functions.
- New data and functions can be easily added as and when required.
- Follows a bottom-up approach for problem solving.

2.1.1 Procedural Programming Verses Object - Oriented Programming

Q2. What is the Difference Between Procedure Oriented Programming (POP) & Object Oriented Programming (OOP).

Ans :

Procedure Oriented Programming	Object Oriented Programming
In POP, program is divided into small parts called functions.	In OOP, program is divided into parts called objects.
In POP,Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world.

POP follows Top Down approach.	OOP follows Bottom Up approach.
POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function, it can be kept public or private so we can control the access of data.
POP does not have any proper way for hiding data so it is less secure.	OOP provides Data Hiding so provides more security.
In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

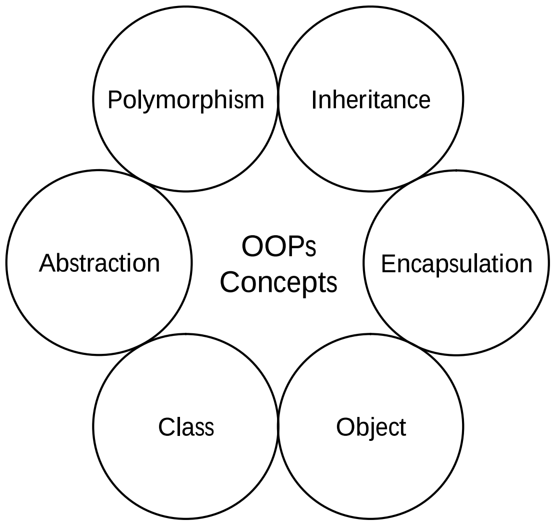
2.1.2 OOPS Terminology

Q3. Explain the basic Concepts of OOPs ?

Ans :

Basic concepts of OOPS

Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.



1. Class

The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user-defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions define the properties and behaviour of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage etc and member functions can apply brakes, increase speed etc.

  We can say that a Class in C++ is a blue-print representing a group of objects which shares some common properties and behaviours.

2. Object

An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

```
classperson
{
   charname[20];
  intid;
public:
  voidgetdetails(){}
};
intmain()
{
  person p1; // p1 is a object
}
```

Object take up space in memory and have an associated address like a record in pascal or structure or union in C.

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

3. Encapsulation

In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsu-lation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales.

Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".

Encapsulation also leads to *data abstraction or hiding*. As using encapsulation also hides the data. In the above example, the data of any of the section like sales, finance or accounts are hidden from any other section.

4. Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

- **Abstraction using Classes:** We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- **Abstraction in Header files:** One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

5. Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

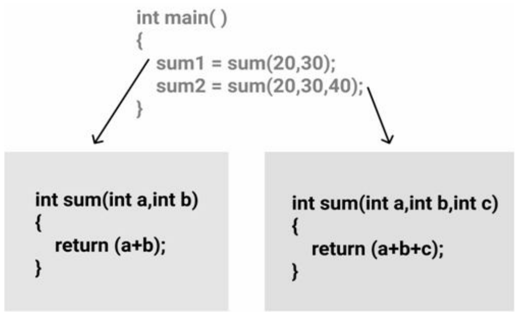
An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

C++ supports operator overloading and function overloading.

- **Operator Overloading:** The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.
- **Function Overloading:** Function overloa-ding is using a single function name to perform different types of tasks.

Polymorphism is extensively used in implementing inheritance.

Example: Suppose we have to write a function to add some integers, some times there are 2 integers, some times there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.

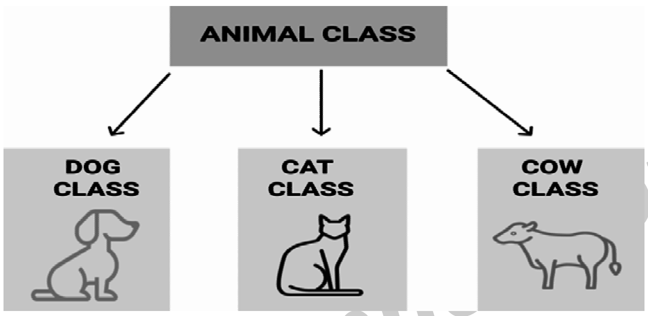


```
int main( )
{
    sum1 = sum(20,30);
    sum2 = sum(20,30,40);
}

int sum(int a,int b)
{
   return (a+b);
}

int sum(int a,int b,int c)
{
   return (a+b+c);
}
```

6. Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class**: The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class: The** class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability**: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.
- **Example**: Dog, Cat, Cow can be Derived Class of Animal Base Class.



7. Dynamic Binding

In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

8. Message Passing

Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

2.1.3 Benefits of Object Oriented Programming

Q4. Explain the Merits and Demerits of object oriented programming.

Ans :

OOP stands for object oriented programming. It offers many benefits to both the developers and the users. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The primary advantages are:

Benefits (or) Merits

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.

- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple objects to coexist without any interference.
- It is possible to map objects in the problem domain to those objects in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in an implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects make the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.
- Polymorphism can be implemented i.e. behavior of functions or operators or objects can be changed depending upon the operations.

Demerits

- It requires more data protection.
- Inadequate for concurrent problems
- Inability to work with existing systems.
- Compile time and run time overhead.
- Unfamiliraity causes training overheads.

2.1.4 OOP Languages

Q5. Explain briefly about object oriented languages.

Ans :

Object-oriented programming is not the right of any particular language. Like structured programming, OOP concepts can be implemented using languages such as C and Pascal. However, programming becomes clumsy and may generate confusion when the programs grow large. A language that is specially designed to support the OOP concepts makes it easier to implement them.

The languages should support several of the OOP concepts to claim that they are object- oriented. Depending upon the features they support, they can be classified into the following two categories:

1. Object-based programming languages, and
2. Object-oriented programming languages.

Object-based programming is the style of programming that primarily supports encapsulation and object identity. Major features that are required for object-based programming are:

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

Languages that support programming with objects are said to be object-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object- based programming language.

Object-oriented programming incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statement:

Object-based features + inheritance + dynamic binding

Languages that support these features include C++, Smalltalk, Object Pascal and Java. There are a large number of object-based and object-oriented programming languages. Table 1.1 lists some popular general purpose OOP languages and their characteristics.

Characteristics	**Simuta ***	**Smalltalk ***	**Objective C**	**C++**	**Ada ****	**Object Pascal**	**Turbo Pascal**	**Eiffel ***	**Java ***
Binding	Both	Late	Both	Both	Early	Late	Early	Early	Both
(early or late)	✓	✓	✓	✓	✓	✓	✓	✓	✓
Polymorphism	✓	✓	✓	✓	✓	✓	✓	✓	✓
Data hiding	✓	✓	✓	✓	✓	✓	✓	✓	✓
Concurrency	✓	Poor	Poor	Poor	Difficult	No	No	Promised	✓
Inheritance	✓	✓	✓	✓	No	✓	✓	✓	✓
Multiple Inheritance	No	✓	✓	✓	No	---	---	✓	No
Garbaga Collection	✓	✓	✓	✓	No	✓	✓	✓	✓
Persistence	No	Promised	No	No	Like 3GL	No	No	Same Support	✓
Genericity	No	No	No	✓	✓	No	No	✓	No
Object Labraries	✓	✓	✓	✓	Not Much	✓	✓	✓	✓

➢ Pure object-oriented languages

➢ Object-based languages

➢ Others are extended conventional languages

Use of a particular language depends on characteristics and requirements of an application, organizational impact of the choice, and reuse of the existing programs. C++ has now become the most successful, practical, general purpose OOP language, and is widely used in industry today.

2.1.5 OOP Applications

Q6. List a few areas of applications of OOP Technology.

Ans :

OOP has become one of the programming buzzwords today. There appears to be a great deal excitement and interest among software engineers in using OOP. Applications of OOP are beginning to gain importance in many areas. The most popular application of object- oriented programming, up to now, has been in the area of user interface design such as windows. Hundreds of windowing systems have been developed, using the OOP techniques.

Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of applications because it can simplify a complex problem. The promising areas for application of OOP include:

- ➢ Real-time systems
- ➢ Simulation and modeling
- ➢ Object-oriented databases
- ➢ Hypertext, hypermedia and expertext
- ➢ AI and expert systems
- ➢ Neural networks and parallel programming
- ➢ Decision support and office automation systems
- ➢ CIM/CAM/CAD systems

The richness of OOP environment has enabled the software industry to improve not only the quality of software systems but also its productivity. Object-oriented technology is certainly, changing the way the software engineers think, analyze, design and implement systems.

2.2 Classes

2.2.1 Introduction

Q7. What is a Class ? Explain with an example.

Ans :

A class is an abstract data type that groups the data and its associated functions. It can also hide the data and functions if required.

A class specification consists of two parts,

(i) Declaration of a class and

(ii) Definitions of member functions of a class.

The members scope and types are described by the class declaration. And, the implementation of member functions are described by the class function definitions.

General Form

```
class classname
{
access_specifier 1 :mem_var1;
⋮
mem_func1;
⋮
access_specifier2 :mem_var2;
⋮
mem_func2;
}
```

The class keyword indicates an abstract data type called name of a class and body of a class includes member variables and member function declarations.

Example

```
// Class Name
class circle
{
// Member Variable Declaration
    private:
        int radius;
        // Member Function Declarations
            public:
        void setdata();
        void getdata(int);
        void display();
        void area();
        void perimeter();
};
```

In the above example, class is a keyword and circle is the name of the class. The radius is a member variable that is declared in private section and member functions are declared as public. The private and public are the access modifiers to provide access to the class members and are terminated by colon (:). The class is enclosed within curly braces and terminated by using a semicolon(;).

The variables that are declared in the private section are accessed only within the class or by the public member functions of the class. The methods that are available in the public section are accessible inside or outside the class by using objects.

The private variables of the class are directly accessible like public variables. This can be done by storing public and private variables in consecutive memory locations. The address of first member variable is stored in a pointer and pointer is incremented or decremented to access both private and public members directly.

When an object is created to the class, the address of variable in first memory location is stored in that object. This object is also used to access private and public members directly.

Q8. How classes provide data encapsulation.

Ans :

A class is an abstract data type that groups the data and its associated functions. It can also hide the data and functions if required.

A class specification consists of two parts.

(i) Declaration of a class and

(ii) Definitions of member functions of a class.

The members scope and types are described by the class declaration. And, the implementation of member functions are described by the class function definitions.

Encapsulation is a mechanism of binding data members and corresponding methods into a single module classed class, inorder to protect them from being accessed by the outside code. An instance of a class can be called as an object and it is used to access the members of a class. In encapsulation, objects are treated as 'block boxes' since each object performs specific task.

The data and functions available in a class are called as members of a class. The data defined in the class are called as member variables or data members and the functions defined are called as member functions.

The main idea behind the concept of encapsulation is to obtain high maintenance and to handle the application's code.

2.2.2 Defining an Instance of a Class

Q9. Define an object. How to define an object in C++?

Ans :

Class is mere a blueprint or a template. No storage is assigned when we define a class. Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects.

Each object has different data variables. Objects are initialised using special class functions called **Constructors**.

Syntax to Define Object in C++

className objectVariableName;

You can create objects of *Test* class (defined in above example) as follows:

```
classTest
{
private:
int data1;
float data2;
public:
```

```
void function1()
{   data1 =2;}

float function2()
{
        data2 =3.5;
return data2;
}
};

int main()
{
Test o1, o2;
}
```

Here, two objects o1 and o2 of Test class are created.

In the above class Test, data1 and data2 are data members and function1() and function2() are member functions.

Q10. Write a program to enter student details and display it using classes.

Ans :

```
#include<iostream>
usingnamespace std;
class stud
{
public:
char name[30],clas[10];
int rol,age;
void enter()
{
     cout<<"Enter Student Name: ";
     cin>>name;
     cout<<"Enter Student Age: ";
     cin>>age;
     cout<<"Enter Student Roll number: ";
     cin>>rol;
     cout<<"Enter Student Class: ";
     cin>>clas;
}
void display()
{
cout<<"\n Age\tName\tR.No.\tClass";
cout<< "\n" <<age<<"\t"<<name<<"\t"<<rol <<"\t"< <clas;
}};
```

```
int main()
{
      class stud s;
      s.enter();
      s.display();
      cin.get();//use this to wait for a keypress
}
```

Q11. Explain, How to access data member and member function in C++?

Ans :

You can access the data members and member functions by using a . (dot) operator. For example,

```
      o2.function1();
```

This will call the function1() function inside the Test class for objects o2.

Similarly, the data member can be accessed as:

```
      o1.data2 = 5.5;
```

It is important to note that, the private members can be accessed only from inside the class.

So, you can use o2.function1(); from any function or class in the above example. However, the code o1.data2 = 5.5; should always be inside the class *Test*.

Q12. What are Member Functions ? Explain about them how to use in class.

Ans :

Member functions are the functions, which have their declaration inside the class definition and works on the data members of the class. The definition of member functions can be inside or outside the definition of class.

If the member function is defined inside the class definition it can be defined directly, but if its defined outside the class, then we have to use the scope resolution :: operator along with class name along with function name.

```
// defining member function
class Cube
{
 public:
 int side;
 int getVolume();
      // Declaring function getVolume with no argument //and return type int.
};
```

If we define the function inside class then we don't not need to declare it first, we can directly define the function.

```
      // another way of defining member function
class Cube
{
 public:
```

```
 int side;
 int getVolume()
 {
  return side*side*side;  //returns volume of cube
 }     };
```

But if we plan to define the member function outside the class definition then we must declare the function inside class definition and then define it outside.

```
// declaring and using member function
class Cube
{
 public:
 int side;
 int getVolume();
}
int Cube :: getVolume()  // defined outside class definition
{
 return side*side*side;
}
```

The main() function for both the function definition will be same. Inside main() we will create object of class, and will call the member function using dot . operator.

```
// calling member function inside main using dot // operator
int main()
{
 Cube C1;
 C1.side=4;          // setting side value
 cout<< "Volume of cube C1 ="<< C1.getVolume();
}
```

Similarly we can define the getter and setter functions to access private data members, inside or outside the class definition.

Q13. Write about various access controls/ modifiers in C++

Ans :

Access Control in Classes

Now before studying how to define class and its objects, lets first quickly learn what are access specifiers.

Access specifiers in C++ class defines the access control rules. C++ has 3 new keywords introduced, namely,

1. public
2. private
3. protected

These access specifiers are used to set boundaries for availability of members of class be it data members or member functions

Access specifiers in the program, are followed by a colon. You can use either one, two or all 3 specifiers in the same class to set different boundaries for different class members. They change the boundary for all the declarations that follow them.

1. Public

Public, means all the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. Hence there are chances that they might change them. So the key members must not be declared public.

```
class PublicAccess
{
 public:          // public access specifier
 int x;           // Data Member Declaration
 void display();  // Member Function declaration
}
```

2. Private

Private keyword, means that no one can access the class members declared private outside that class. If someone tries to access the private member, they will get a compile time error. By default class variables and member functions are private.

```
class PrivateAccess
{
 private:     // private access specifier
 int x;           // Data Member Declaration
 void display();  // Member Function declaration
}
```

3. Protected

Protected, is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class. (If class A is inherited by class B, then class B is subclass of class A. We will learn this later.)

```
class ProtectedAccess
{
 protected:       // protected access specifier
 int x;           // Data Member Declaration
 void display();  // Member Function declaration
}
```

Q14. How to access public data members in C++.

Ans :

Accessing Public Data Members

Following is an example to show you how to initialize and use the public data members using the dot (.) operator and the respective object of class.

```
// demonstration of accessing public data members
class Student
{
 public:
 int rollno;
 string name;
};
int main()
{
 Student A;
 Student B;
 A.rollno=1;
 A.name=”Adam”;
 B.rollno=2;
 B.name=”Bella”;
      cout <<“Name and Roll no of A is :”<<  A.name << A.rollno;
      cout <<“Name and Roll no of B is :”<<  B.name << B.rollno;
}
```

2.3 WHY HAVE PRIVATE MEMBERS?

Q15. How to access private data members in C++ with an example.

Ans :

Accessing Private Data Members

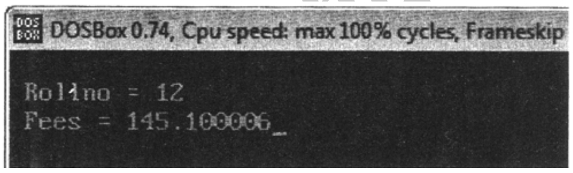
To access, use and initialize the private data member you need to create getter and setter functions, to get and set the value of the data member.

The setter function will set the value passed as argument to the private data member, and the getter function will return the value of the private data member to be used. Both getter and setter function must be defined public.

Example :

```
include<iostream.h>
#include<conio.h>
class student
{
      private:
      int rollno;
      float fees;
      void read()
      {
              rollno=12;
              fees=145.10;
      }
      public:
      void show()
      {
              read();
              cout<<"\n Rollno ="<<rollno;
              cout<<"\n Fees ="<<fees;
      }
};
void main()
{
      clrscr();
      student st;
      //st.read();          // not accessible
      st.show ();
      getch();
}
```

Output


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip

```
Rollno = 12
Fees = 145.100006_
```

Q16. What is the need for private members?

Ans :

The classes contain both the functions and variables that are to be used in the program. Sometimes the class members might need to be used only within the class internally. It means the statements which are outside the class should not have access to these members. With this, the critical data can be protected from being modified.

The solution for this is to make the members private. This can be done by preceding the class members with private keyword. When a variable is declared as private, the values can be only stored in it through public member function. This function is the only means for the application to access the private variables.

2.4 Separating Class Specification from Implementation

Q17. How to separate a class specification from implementation? Explain with the help of an example.

Ans :

Let us consider the following example

```
//A very simple class
#include <iostream>
using namespace std;
//Rectangle class declaration
class Rectangle {
private:
double width;
double length;
public:
Rectangle();                   //constructor
Rectangle(double, double); //constructor
void setWidth(double);
void setLength(double);
double getWidth() const;
double getLength() const;
double area() const;
};
Rectangle::Rectangle() {
width=1;
length=1;
} //constructor
Rectangle::Rectangle(double w, double l) {
width=w;
length=l;
} //constructor
void Rectangle::setWidth(double w) {
width=w;
} //setWidth
void Rectangle::setLength(double l) {
length=l;
} //setLength
```

```
double Rectangle::getWidth() const {
return width;
} //getWidth
double Rectangle:: getLength() const {
return length;
} //getWidth
double Rectangle::area() const {
return width*length;
} //getArea
int main() {
Rectangle box;   //box is an instance of Rectangle class
box.setWidth(5);//set member attribute width
box.setLength(8);       //set member attribute length
Rectangle crate;  //box is an instance of Rectangle
                  //class 1x1
Rectangle carton(3,6); //carton is declared using           //constructor 3x6
cout<<"Area of box is "<<box.area()<<endl;
                        //prints 40
cout<<"Area of carton is "<<carton.area()<<endl;
                        //prints 18
cout<<"Area of crate is "<<crate.area()<<endl;
                        //prints 1
system("pause");
return 0;
}                       //main
```

Previously, we had combined all of the code for the class and main into one cpp file.

We will now separate the code into 3 separate files: Rectangle.h, Rectangle.cpp and main.cpp (or whatever name you want to give the main program.)

Rectangle.h is the Class Specification

```
//Specification File for the Rectangle class
#ifndef RECTANGLE_H
#define RECTANGLE_H
#include <string>
#include <sstream>
using namespace std;
class Rectangle {
private:
double width;
double length;
public:
Rectangle(void);
Rectangle(double, double); //constructor
void setWidth(double);
void setLength(double);
double getWidth() const;
double getLength() const;
double perimeter() const;
double diagonal() const;
double area() const;
bool square() const;
void increase(); //increase length and width by one
void increase(double); //increase length and width
by amount
string toString() const;
};
#endif
```

Rectangle.cpp is the Class Implementation:

```
//Implementation File for the Rectangle class
#include "Rectangle.h"
Rectangle::Rectangle() {
width=1;
length=1;
} //constructor
Rectangle::Rectangle(double w, double l) {
if(w>0) width=w;
if(l>0) length=l;
} //constructor
void Rectangle::setWidth(double w) {
if(w>0) width=w;
} //setWidth
void Rectangle::setLength(double l) {
if(l>0) length=l;
} //setLength
double Rectangle::getWidth() const {
return width;
} //getWidth
void Rectangle:: increase() {
width++;
length++;
}
void Rectangle:: increase(double amount) {
if(amount>0) {
width+=amount;
length+=amount;
}
}
double Rectangle:: getLength() const {
return length;
} //getWidth
double Rectangle::perimeter() const {
return (width+length)*2;
}
double Rectangle::diagonal() const {
return sqrt(width*width+length*length);
}
double Rectangle::area() const {
return width*length;
} //getArea
bool Rectangle::square() const {
return width==length;
} //square
string Rectangle::toString() const {
std::ostringstream oss; //a stream to append the
values
oss<<width<<"X"<<length;
return oss.str();
} //toString
#include <iostream>
#include <string>
#include "Rectangle.h"
using namespace std;
void main() {
Rectangle myBox;
double wdth, lnth;
cout<<"Enter width of rectangle:";
cin>>wdth;
myBox.setWidth(wdth);
if(myBox.getWidth()!=wdth) cout<<"The value
"<<wdth<<" was not accepted\n";
cout<<"Enter length of rectangle:";
cin>>lnth;
myBox.setLength(lnth);
if(myBox.getLength()!=lnth) cout<<"The value "<<lnth<<" was not accepted\n";
cout<<myBox.toString()<<endl;
cout<<"Area of mybox is "<<myBox.area()<<endl; //prints 40
if(myBox.square()) cout<<"MyBox is a square"<<endl; //prints "The box is a square"
cout<<"The perimeter is "<<myBox.perimeter()<<endl;
cout<<"The diagonal is "<<myBox.diagonal()<<endl;
myBox.increase(5.3);
cout<<myBox.toString()<<endl;
system("pause");
}
```

2.5 Inline Member Functions

Q18. Explain about Inline member functions.

Ans :

A member function that is both declared and defined in the class member list is called an *inline member function*. Member functions containing a few lines of code are usually declared inline.

An equivalent way to declare an inline member function is to declare it outside of the class declaration using the keyword inline and the :: (scope resolution) operator to identify the class the member function belongs to. For example:

```
class Y
{
    char* a;
public:
    char* f() {return a;};
};
```

is equivalent to:

```
class Z
{
    char* a;
public:
    char* f();
};
inline char* Z::f() {return a;}
```

When you declare an inline function without the inline keyword and do not define it in the class member list, you cannot call the function before you define it. In the above example, you cannot call f() until after its definition.

Inline member functions have internal linkage. Noninline member functions have external linkage.

To write a program to find the multiplication values and the cubic values using inline function.

```
#include<iostream.h>
#include<conio.h>
 class line
{
   public:
            inline float mul(float x,float y)
            {
                        return(x*y);
            }
            inline float cube(float x)
            {
                        return(x*x*x);
            }
};
void main()
{
            line obj;
            float val1,val2;
            clrscr();
            cout<<"Enter two values:";
            cin>>val1>>val2;
         cout<<"\nMultiplication value

is:"<<obj.mul(val1,val2);
         cout<<"\n\nCube value is :"<<obj.cube
               (val1)<<"\t"<<obj.cube(val2);
            getch();
}
```

Output:

```
Enter two values: 5   7
Multiplication Value is: 35
Cube Value is: 25 and 343
```

2.6 Constructors

Q19. What are constructors ? Explain them with syntax and example.

Ans :

Constructors

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors iitialize values to object members after storage is allocated to the object.

Syntax:

```
class A
{
 int x;
 public:
 A();        //Constructor
};
```

While defining a constructor you must remember that the name of constructor will be same as the name of the class, and constructors never have return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

Example of constructor

```
class A
{
 int i;
 public:
 A(); //Constructor declared
};
A::A()   // Constructor definition
{
 i=1;
}
```

Q20. Explain the types of constructors.

Ans :

Types of Constructors

Constructors are of three types

1. Default Constructor
2. Parametrized Constructor
3. Copy Constructor

1. **Default Constructor**

   Default constructor is the constructor which doesn't take any argument. It has no parameter.

2. **Parameterised Constructors**

   These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

3. **Copy Constructor**

   Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type. It is usually of the form X (X &), where X is the class name.he compiler provides a default Copy Constructor to all the classes.

Q21. Explain briefly about default constructor.

Ans :

Default Constructor

Default constructor is the constructor which doesn't take any argument. It has no parameter.

Syntax :

```
class_name ()
{ Constructor Definition }
```

Example :

```
class Cube
{
int side;
public:
Cube()
 {
  side=10;
 }
};
int main()
{
Cube c;
cout << c.side;
}
```

Output : 10

In this case, as soon as the object is created the constructor is called which initializes its data members.

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

2.6.1 Passing Arguments to Constructors

Q22. How to pass arguments to constructors? Explain with an example

Or

Explain briefly about parameterised Constructors

Ans :

Arguments can be passed to constructors by defining the parameterised constructors.

Parameterised Constructors

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

```
// pararmeterised constructor :
class Cube
{
 int side;
 public:
 Cube(int x)
  {
   side=x;
  }
};

int main()
{
 Cube c1(10);
 Cube c2(20);
 Cube c3(30);
 cout << c1.side;
 cout << c2.side;
 cout << c3.side;
}
```

OUTPUT : 10 20 30

By using parameterized constructor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

Q23. Explain briefly about copy constructor.

Ans :

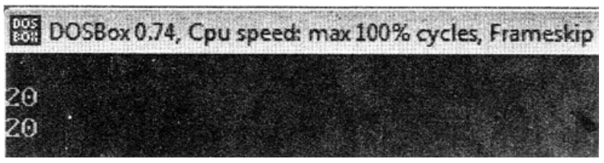
Copy constructor is a constructor function used for copying objects. It has the same name as that of a class and takes a reference to a constant parameter.

This constructor copies one object from the other sequentially during the object declaration. This process of initializing is called copy initialization.

Program

```
#include<iostream.h>
#include<conio.h>
class Example
{
    private:
        int data;
    public:
    Example()
    {
    }
    Example(int a)
    {
        data = a;
    }
    Example(Example &c)
    {
        data = c.data;
        cout<<"Copy constructor invoked";
    }
    void show()
    {
        cout<<data;
    }
};
main()
{
    clrscr();
        Example e(20);
        e.show();
        Example el;
        el = e;
        el.show();
        getch();
        return 0;
    }
```

Output



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip

```
20
20
```

2.6.2 Destructors

Q24. What is the use of Destructors ? Write its syntax and explain.

Ans :

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a **tilde** ~ sign as prefix to it.

Syntax:

```
class A
{
 public:
  ~A();
};
```

Destructors will never have any arguments.

Example to see how Constructor and Destructor is called

```
class A
{
A()
 {
  cout << "Constructor called";
 }
~A()
 {
  cout << "Destructor called";
 }
};

int main()
{
 A obj1;   // Constructor Called
 int x=1
 if(x)
  {
   A obj2;  // Constructor Called
  }   // Destructor Called for obj2
} //  Destructor called for obj1
```

Q25. How to allocate and deallocate memory for classes and objects in C++.

Or

Explain new and delete operators in C++.

Ans :

Memory allocations for classes and objects

Arrays can be used to store multiple homogenous data but there are serious drawbacks of using arrays. Programmer should allocate the memory of an array when they declare it but most of time, the exact memory needed cannot be determined until runtime. The best thing to do in this situation is to declare the array with maximum possible memory required (declare array with maximum possible size expected) but this wastes memory.

To avoid wastage of memory, you can dynamically allocate the memory required during runtime using new and delete operator.

The new Operator

Syntax : ptr = new float[n];

This expression in the above program returns a pointer to a section of memory just large enough to hold the n number of floating-point data.

The delete Operator

Once the memory is allocated using new operator, it should released to the operating system. If the program uses large amount of memory using new, system may crash because there will be no memory available for operating system. The following expression returns memory to the operating system.

Syntax : delete [] ptr;

The brackets [] indicates that, array is deleted. If you need to delete a single object then, you don't need to use brackets.

Example : delete ptr;

```
  // demonstration of new and delete operators
#include<iostream>
usingnamespace std;
classTest{
private:
int n;
float*ptr;
public:
Test(){
        cout <<"Enter total number of students: ";
        cin >> n;
        ptr =newfloat[n];
        cout <<"Enter GPA of students."<<endl;
for(int i =0; i < n;++i){
            cout <<"Student"<< i+1<<": ";
            cin >>*(ptr + i);
}
}
~Test(){
delete[] ptr;
}
voidDisplay(){
        cout <<"\nDisplaying GPA of students."<<
endl;
for(int i =0; i < n;++i){
            cout <<"Student"<< i+1<<" :"<<*(ptr
+ i)<< endl;
}
}
};
```

```
int main(){
Test s;
   s.Display();
return0;
}
```

The output of this program is same as above program. When the object s is created, the constructor is called which allocates the memory for n floating-point data.

When the object is destroyed, that is, object goes out of scope then, destructor is automatically called.

```
   ~Test() {
      delete[] ptr;
   }
```

This destructor executes delete[] ptr; and returns memory to the operating system.

Passing and Returning Object from Function in C++ Programming

In C++ programming, objects can be passed to function in similar way as variables and structures.

2.6.3 Overloading Constructors

Q26. What is Constructor Overloading?

Ans :

Just like other member functions, constructors can also be overloaded. Infact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other with parameter.

You can have any number of Constructors in a class that differ in parameter list.

```
// demonstration of overloading constructor
class Student
{
 int rollno;
 string name;
 public:
 Student(int x)
 {
  rollno=x;
  name="None";
 }
 Student(int x, string str)
 {
  rollno=x ;
  name=str ;
 }
};
int main()
{
 Student A(10);
 Student B(11,"Ram");
}
```

In above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

In the above case if we write Student S; in **main()**, it will lead to a compile time error, because we haven't defined default constructor, and compiler will not provide its default constructor because we have defined other parameterized constructors.

Q27. Write a C++ Program To Overload The Constructor.

Ans :

```
#include <iostream.h>
#include<conio.h>
class MyClass
{
        public:
        int x;
          int y;
        // Overload the default constructor.
                        MyClass()
                {
                      x = y = 0;
                }
                  // Constructor with one parameter.
                      MyClass(int i)
                {
                    x=y= i;
                }
                    // Constructor with two parameters.
                  MyClass(int i, int j)
              {
                  x=i;
                  y=j;
              }
};
        void main()
      {
                clrscr();
        MyClass t; // invoke default constructor
        MyClass t1 (5); // use MyClass(int)
        MyClass t2(9, 10); // use MyClass(int, int)
        cout<< "t.x: " << t.x<< ", t.y: "
                        <<t.y<< "\n";
        cout << "t1.x: " << t1.x << ", t1.y: "
                        << t1.y << "\n";
        cout << "t2.x: " << t2.x << ", t2.y: "
                        <<t2.y << "\n";
                getch();
        }
```

Q28. What is this Pointer? Explain with an example

Ans :

Every object in C++ has access to its own address through an important pointer called this pointer. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a this pointer, because friends are not members of a class. Only member functions have a this pointer.

```
// demonstration of this pointer
#include <iostream>
using namespace std;
class Box
{
  public:
    // Constructor definition
    Box(double l=2.0, double b=2.0, double h=2.0)
    {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
    }
    double Volume()
    {
      return length * breadth * height;
    }
    int compare(Box box)
    {
       return this->Volume() > box.Volume();
    }
  private:
    double length;          // Length of a box
    double breadth;   // Breadth of a box
    double height;          // Height of a box
};
int main(void)
{
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2
   if(Box1.compare(Box2))
  {
    cout << "Box2 is smaller than Box1" <<endl;
  }
  else
  {
   cout << "Box2 is equal to or larger than Box1"
<<endl;
  }
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Output

```
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
```

2.7 Private Member Functions

Q29. What are private member functions? Write a program to demonstrate it.

Ans :

Private Member Functions

Usually member data are made private while functions (or methods) are made public. There might be instances where you might want to make certain functions private (i.e. you may not want the user to directly access these functions). Private functions can only be called from within public member functions. These functions are also called 'helper functions' Why do we need them?

Let's take the example of the class 'batsman'. After every match the user will enter the batsman's new score and then he will have to call two functions to update the batsman's record (i.e. the user has to call update_best ( ) and update_worst ( )). It is unnecessary to bother the user with this kind of a double function call. Why should the user access these two functions directly? Instead of this, we could define another function called update ( ) which will call update_best ( ) and update_worst ( ). In this way the user needs to only call one function after every match.

The idea of classes is to restrict user access. We don't want the user to access data or functions unnecessarily. So, we will make update_best ( ) and update_worst ( ) as private functions while update ( ) will be a public function.

Example

```
#include <iostream.h>
class batsman
{
private:
int player_number;
int best_score,worst_score;
void update_best(int);
void update_worst(int);
public:
batsman(int n, int b, int w) //constructor
{
player_number=n;
best_score=b;
worst_score=w;
}
void update(int);
void display( );
};
void batsman::update(int x)
{
update_best(x); //private function is called
update_worst(x);
cout<<"\n\nThe scores have been updated\n";
}
void batsman::display( )
{
cout<<"\nHighest score : "<<best_score;
cout<<"\nLowest score : "<<worst_score;
}
void batsman::update_best(int y)
            //defining the private functions
{
    if (y>best_score)
    {
    best_score=y;
    }
}
void batsman::update_worst(int z)
{
    if (z<worst_score)
    {
    worst_score=z;
    }
}
int main( )
{
batsman b(1, 140, 20);
cout<<"The scores before the match is ";
b.display( );
b.update(180);
cout<<"\nAfter this match the scores are ";
b.display( );
return 0;
}
```

```
The output will be:
The scores before the match is
Highest score : 140
Lowest score : 20
The scores have been updated
After this match the scores are
Highest score : 180
Lowest score : 20
```

2.8 Arrays of Objects

Q30. Explain about creating and accessing of Arrays of Objects with an example.

Ans :

Arrays of Objects

An array can be of any data type including struct. Similarly, we can also have arrays of variables that are of the type class. Such variables are called arrays of Objects. Consider the following class definition:

```
class employee
{
        char name[30];
        float age;
    public:
        void getdata(void);
        void putdata(void);
};
```

The identifier employee is a user-defined data type and can be used to create objects that relate to different categories of the employees.

Example:

```
employee manager [3];        // array of manager
employee foreman [15];       // array of foreman
employee worker [75];        // array of worker
```

The array manager contains three objects(managers), namely, manager[0], manager [1] and manager [2], of type employee class. Similarly, the foreman array contains 15 objects (foremen) and the worker array contains 75 objects(workers).

Since an array of objects behaves like any other array, we can use the usual array accessing methods to access individual elements, and then the dot member operator to access the member functions. For example, the statement

```
manager[i].putdata();
```

will display the data of the ith element of the array manager. That is, this statement requests the object manager[i] to invoke the member function putdata().

An array of objects is stored inside the memory in the same way as a multi-dimensional array. The array manager is represented in Fig. Note that only the space for data items of the objects is created. Member functions are stored separately and will be used by all the objects.

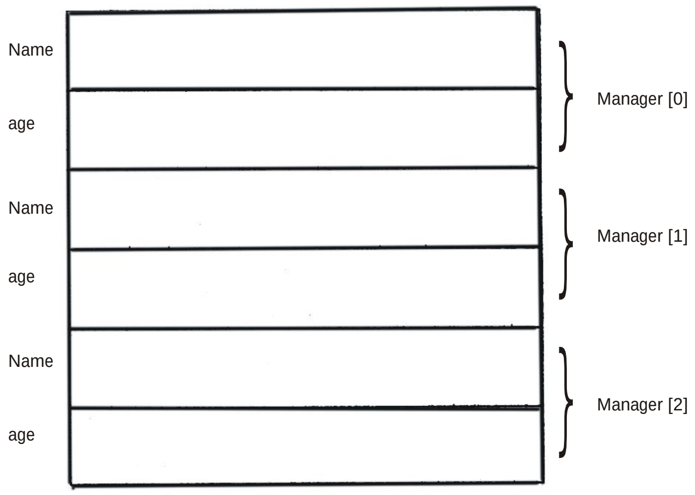


Fig. : Storage of data items of an object array

Example :

```
        # Iinclude <iostream>
        using namespace std;
        class employee
{
        char  name[30];       //  string as class member
        float age;
public:
        void getdata(vcid),
        void putdata(void);
};
void employee :: getdata(void)
{
        cout << "Enter name:  " ;
        cin >> name: ;
        cout << "Enter age: ";
        cin >> age;
}
void employee :: putdata(void)
{
        cout << "Name: " << name << "\n";
        cout << "Age: " <<  age << "\n";
}
const int size=3;
```

```
        int main()
        {
                employee manager[size];
                for(int i=0; i<size; i++)
        {
                cout << "\nDetails of manager" << i+1 << "\n"; manager[i].getdata();
        }
                cout << "\n";
                for(i=0; i<size; i++)
        {
        cout<<"\nManager"<< i+1<< "\n"; manager[i].putdata ( );
        }
        return 0;
}
```

Output :

```
        Interactive Input
        Details of managerl
        Enter name  : xxx
        Enter age : 45
        Details of manager2
        Enter name: yyy Enter
        age: 37
        Details of manager3
        Enter name: zz
        Enter age: 50
        Program output
        Manager1
        Name: Raki
        Age: 45
        Manager2
        Name: Raju
        Age: 37
        Manager3
        Name: Ramesh
        Age: 50
```

2.9 Instance and Static Members

Q31. What are instance member of a class ? Define them with an examples.

Ans :

Instance Members of a Class

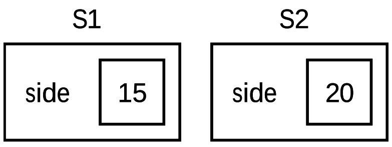
A class contains several objects that are considered as instances of that class. Every object maintains a copy of the class member variables for itself. Similarly, even other objects maintain copies of the class member variables. The member variables of an object are different from the member variables of the other object of the same class. Consider a class square that has two objects SI and S2 defined for it.

Let the side of the square set as shown below,

S1.setside(15);

S2.setside(20);

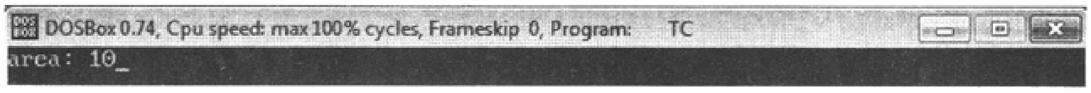
Here, S1 and S2 are two different objects having its own dimension called side.



Example

```
#include <iostream.h>
#include<conio.h>
class Rectangle
{
    int width, height;
    public:
    void setvalue(int,int);
    int area()
    {
        return width*height;
    }
};
void Rectangle::setvalue(int x, int y)
{
    width = x;
    height = y;
}
int main()
{
    Rectangle r;
    clrscr();
    r.setvalue(2,5);
    cout«”area:”«r.area();
    getch();
    return 0;
}
```

Output


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
area: 10_
```

Q32. Write about static members of a class.

Ans :

Static Members of a Class

The static members of a class are static member function and static member variable.

(i) Static Member Function

When a member function is preceded by static keyword then it is called as static member function. The static member functions have ability to access static member variables and the member functions of the same class. These static member functions are invoked by the class name without using the class object. The scope of static member is valid in the entire class, but, it does not create any side effects to other part of the program.

There are few important points to remember while using static member functions. They are as follows,

1. For static members, memory is allocated only once to entire class and all the class members will share the copy of the memory.
2. The static member function is invoked by class name, followed by function name and terminated with semicolon.

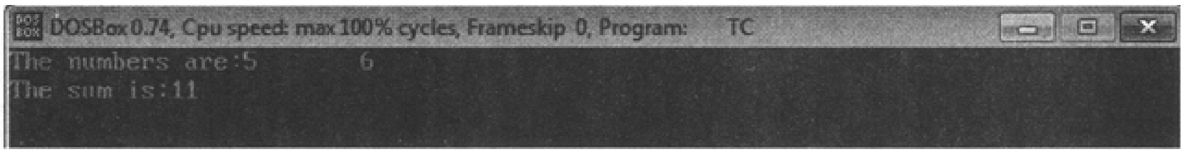
   The class name and function name are separated by scope resolution operator (::).
3. The static member functions are also invoked by using objects.

Example

```
#include<iostream.h>
#include<conio.h>
class number
{
        private:
        static int result; //static member variable declaration
        static int a, b; //declaration of static member variables
        public:
        static void add() //definition of static member function
        {
                result=a+b;
        }
        static void display()
        {
                cout<<"The numbers are:"<<a<<"\t"<<b<<endl;
                cout<<"The sum is:"<<result;
        }
};
int number: :result=0;
int number::a=5;
int number: :b=6;
int main()
```

```
{
        clrscr();
        number: :add( );        //calls add function
        number: :display();     //calls display function
        getch();
        return 0;
}
```

Output


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
The numbers are:5        6
The sum is:11
```

(ii) Static Member Variable

A member variable that is preceded with the static keyword is called as static member variable. In a class, memory is allocated to all the objects and member variables are assigned to all objects. There is a possibility for member variables to allocate memory like member functions. This can be done by using the keyword static.

The static variables that are declared in the class have limited access within the class, but they are alive till the execution of the program is completed. When a local variable is declared as static it maintains last variable's value.

Syntax

static<definition_of_variable>

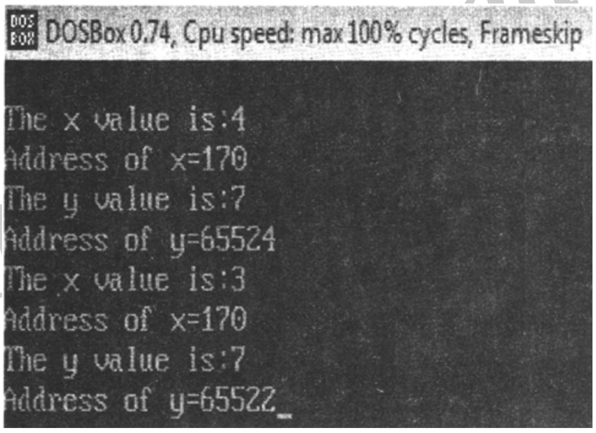
Some of the reasons to declare static member variables outside the class are as follows,

1. The memory is allocated separately to static member variables irrespective of objects.
2. The static member variables must be initialized with a value to avoid linker errors.
3. For the static member variables, memory is allocated at most one time.
4. The memory is allocated only once for static variables and the entire class objects will use the common static variable.

Program

```
#include<iostream.h>
#include<conio.h>
class example
{
        static int x;           //  declaration of static variable
        int y;                  //  declaration of non-static variable
        public:void null()
        {
                y=8;
        }
        void total()
        {
                x—;
                y—;
                cout<<“\n The x value is:”<<x<<“\t”<<“\n Address of x=”<<(unsigned)&x;
                cout<<“\n The y value is:”<<y<<“\t”<<“\n Address of y=”<<(unsigned)&y;
        }
};
int example: :x=5; //initialization of static member variable
int main()
{
        example exl, ex2;
        ex1.null();
        ex2.null();
        ex1.totalf);
        ex2.total();
        getch();
        return 0;
}
```

Output



```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip

The x value is:4
Address of x=170
The y value is:7
Address of y=65524
The x value is:3
Address of x=170
The y value is:7
Address of y=65522_
```

1. A friend function is not a member function of a class to which it is declared as friend. Since, this friend function is out of scope of that particular class. Therefore, the objects of that class can’t be used to call this friend function. These functions are invoked directly.
2. It can be declared either in public or private section of a class.
3. It can be invoked with out any object.
4. It is used in operator overloading.
5. It is called like any other normal function.
6. It can be called without including the dot(.) operator.
7. It takes objects of class type as arguments.
8. It is not allowed to access private members of a class. These private members are accessed only through dot and scope resolution operators.

Q33. Write a program to demonstrate static member functions

Ans :

```
#include<iostream>
usingnamespace std;
classBox{
public:
staticint objectCount;
// Constructor definition
Box(double l =2.0,double b =2.0,double h =2.0){
        cout <<"Constructor called."<< endl;
        length = l;
        breadth = b;
        height = h;
// Increase every time object is created
        objectCount++;
}
doubleVolume(){
return length * breadth * height;
}
staticint getCount(){
return objectCount;
}
private:
double length;// Length of a box
double breadth;// Breadth of a box
double height;// Height of a box
};
// Initialize static member of class Box
intBox::objectCount =0;
int main(void){
// Print total number of objects before creating object.
    cout <<"Inital Stage Count:"<<Box::get
              Count()<< endl;
        BoxBox1(3.3,1.2,1.5);// Declare box1
        BoxBox2(8.5,6.0,2.0);// Declare box2
// Print total number of objects after creating object.
        cout <<"Final Stage Count: "<<Box::get
                   Count()<< endl;
return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

2.10 Friends of Classes

Q34. What are friend functions?

Ans :

Friend functions

Friend functions are actually not class member function. Friend functions are made to give private access to non-class functions. You can declare a global function as friend, or a member function of other class as friend.

Hence, friend functions can access private data members by creating object of the class. Similarly we can also make function of other class as friend, or we can also make an entire class as friend class.

When we make a class as friend, all its member functions automatically become friend functions.

Friend Functions is a reason, why C++ is not called as a pure Object Oriented language. Because it violates the concept of Encapsulation.

Q35. Explain the properties of friend function.

Ans :

1. A friend function is not a member function of a class to which it is declared as friend. Since, this friend function is out of scope of that particular class. Therefore, the objects of that class can't be used to call this friend function. These functions are invoked directly.
2. It can be declared either in public or private section of a class.
3. It can be invoked with out any object.
4. It is used in operator overloading.
5. It is called like any other normal function.

6. It can be called without including the dot(.) operator.
7. It takes objects of class type as arguments.
8. It is not allowed to access private members of a class. These private members are accessed only through dot and scope resolution operators.

Q36. How to declare the function as a friend function?

Or

Write the syntax to define friend function

Ans :

Declaration of friend function in C++

```
class class_name
{
   ... .. ...
   friend return_type function_name(argument/s);
   ... .. ...
}
```

Now, you can define the friend function as a normal function to access the data of the class. No friend keyword is used in the definition.

```
class className
{
   ... .. ...
   friend return_type functionName(argument/s);
   ... .. ...
}
return_type functionName(argument/s)
{
   ... .. ...
   // Private and protected data of className can be accessed from
   // this function because it is a friend function of className.
   ... .. ...
}
```

Q37. Write a program for Addition of members of two different classes using friend Function.

Ans :

```
#include<iostream>
usingnamespace std;
// forward declaration
class B;
class A {
private:
int numA;
public:
      A(): numA(12){}
// friend function declaration
friendint add(A, B);
};
class B {
private:
int numB;
public:
       B(): numB(1){}
// friend function declaration
friendint add(A , B);
};
// Function add() is the friend function of classes A and B
// that accesses the member variables numA and numB
int add(A objectA, B objectB)
{
return(objectA.numA  +  objectB.numB);
}
int main()
{
   A objectA;
   B objectB;
    cout<<"Sum: "<< add(objectA, objectB);
return0;
}
```

Output

```
Sum: 13
```

Q38. What is friend class? Write a syntax to define friend class.

Ans :

friend class

Similarly like, friend function. A class can be made a friend of another class using keyword friend.

Syntax: .

```
.......  .....  ........
class A{
  friend class B;      // class B is a friend class
   ..... ..... .....
}
class B{
   ..... ..... .....
}
```

When a class is made a friend class, all the member functions of that class becomes friend function. In this program, all member functions of class B will be friend function of class A. Thus, any member function of class B can access the private and protected data of class A.

If B is declared friend class of A then, all member functions of class B can access private data and protected data of class A but, member functions of class A cannot private and protected data of class B. Remember, friendship relation in C++ is granted not taken.

Q39. Write a program to find the height and width of square and rectangle using friend class.

Ans :

```
#include <iostream>
using namespace std;
class Square;
class Rectangle {
        int width, height;
public:
        Rectangle(int w = 1, int h = 1): width(w), height (h){}
        void display() {
              cout << "Rectangle: " << width * height  << endl;
        };
        void morph(Square &);
};
class Square {
        int side;
public:
        Square(int s = 1):side(s){}
        void display() {
              cout << "Square: " << side * side
                                   << endl;
        };
        friend class Rectangle;
};
void Rectangle::morph(Square &s) {
        width = s.side;
        height = s.side;
}
int main () {
        Rectangle rec(5,10);
        Square sq(5);
        cout << "Before:" << endl;
        rec.display();
        sq.display();
        rec.morph(sq);
        cout << "\nAfter:" << endl;
        rec.display();
        sq.display();
        return 0;
}
```

2.11 Memberwise Assignment

Q40. Explain briefly about memberwise assignment with an example.

Ans :

The methods for default assignment and initialization are "memberwise assignment" and "memberwise initialization," respectively. Memberwise assignment consists of copying one object to the other, a member at a time, as if assigning each member individually. Memberwise initialization consists of copying one object to the other, a member at a time, as if initializing each member individually. The primary difference between the two is that memberwise assignment invokes each member's assignment operator (operator=), whereas memberwise initialization invokes each member's copy constructor.

Memberwise assignment is performed only by the assignment operator declared in the form:

type & type :: operator=([const | volatile] type&)

Default assignment operators for memberwise assignment cannot be generated if any of the following conditions exist:

- A member class has const members.
- A member class has reference members.
- A member class or its base class has a private assignment operator (operator=).
- A base class or member class has no assignment operator (operator=).

Default copy constructors for memberwise initialization cannot be generated if the class or one of its base classes has a private copy constructor or if any of the following conditions exist:

- A member class has const members.
- A member class has reference members.
- A member class or its base class has a private copy constructor.
- A base class or member class has no copy constructor.

The default assignment operators and copy constructors for a given class are always declared, but they are not defined unless both of the following conditions are met:

- The class does not provide a user-defined function for this copy.
- The program requires that the function be present. This requirement exists if an assignment or initialization is encountered that requires memberwise copying or if the address of the class's operator= function is taken.

If both of these conditions are not met, the compiler is not required to generate code for the default assignment operator and copy constructor functions (elimination of such code is an optimization performed by the Microsoft C++ compiler). Specifically, if the class declares a user-defined operator= that takes an argument of type "reference to class-name," no default assignment operator is generated. If the class declares a copy constructor, no default copy constructor is generated.

Therefore, for a given class A, the following declarations are always present:

```
// Implicit declarations of copy constructor
// and assignment operator.
A::A( const A& );
A& A::operator=( const A& );
```

The definitions are supplied only if required (according to the preceding criteria). The copy constructor functions shown in the preceding example are considered public member functions of the class.

Default assignment operators allow objects of a given class to be assigned to objects of a public base-class type. Consider the following code:

Example

```
// spec1_memberwise_assignment_and_initialization.cpp
#include<stdio.h>
class Account
{
protected:
    int _balance;
public:
   int getBalance()
   {
      return _balance;
   }
};
class Checking : public Account
{
private:
    int _fOverdraftProtect;
public:
    Checking(int balance, int fOverdraftProtect)
    {
       _balance = balance;
       _fOverdraftProtect = fOverdraftProtect;
    }
};

int main()
{
    Account account;
    Checking checking(1000, 1);
    account = checking;
    printf_s("Account balance = %d\n", account.getBalance());
}
```

```
Account balance = 1000
```

2.12 Copy Constructor

Q41. What is copy constructor? Explain with the help of syntax.

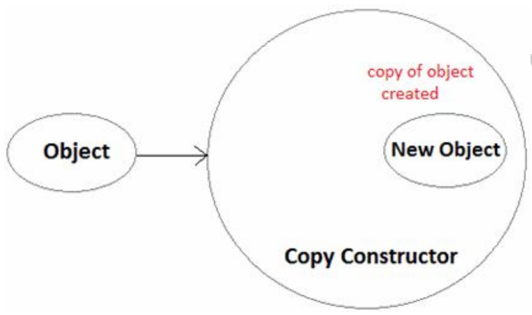
Ans :

Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type. It is usually of the form **X (X&)**, where X is the class name.he compiler provides a default Copy Constructor to all the classes.

Syntax of Copy Constructor

```
class-name (class-name &)
{
 ....
}
```

As it is used to create an object, hence it is called a constructor. And, it creates a new object, which is exact copy of the existing copy, hence it is called **copy constructor**.



Q42. Write a Program to Calculate Prime Number Using Constructor.

Ans :

```
#include<iostream.h>
#include<conio.h>
class prime
{
        int a,k,i;
      public:
      prime(int x)
      {
              a=x;
      }
              void calculate()
              {
                k=1;
                {
                      for(i=2;i<=a/2;i++)

          if(a%i==0)
                      {
                                 k=0;
                                 break;
                      }
                      else
                      {
                               k=1;
                      }
                  }
              }
void show()
              {
                if(k==1)
                cout<< "\n\tA is prime Number. ";
                else
                  cout<<"\n\tA is Not prime.";
              }
};
void main()
{
     clrscr();
     int a;
     cout<<"\n\tEnter the Number:";
     cin>>a;
     prime obj(a);
     obj.calculate();
     obj.show();
     getch();
}
```

Sample Output

Enter the number: 7 Given number is Prime Number.

Q43. Write a program To calculate factorial of a given number using copy constructor.

Ans :

```
#include<iostream.h>
#include<conio.h>
class copy
{
                int var,fact;
                public:
                   copy(int temp)
                   {
                    var = temp;
                   }
                   double calculate()
                   {
                                 fact=1;
                               for(int i=1;i<=var;i++)
                                 {
                                fact = fact * i;
                                 }
                                 return fact;
                   }
};
void main()
{
      clrscr();
      int n;
      cout<<"\n\tEnter the Number : ";
      cin>>n;
      copy obj(n);
      copy cpy=obj;
      cout<<"\n\t"<<n<<" Factorial is:"<<obj.calculate();
      cout<<"\n\t"<<n<<" Factorial is:"<<cpy.calculate();
      getch();
}
```

Output:

```
Enter the Number: 5
Factorial is: 120
Factorial is: 120
```

2.13 Operator Overloading

Q44. Define overloading.

Ans :

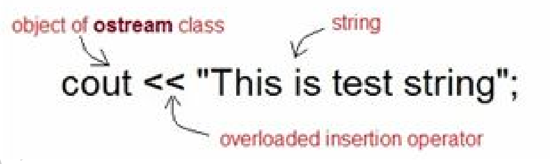
Overloading is one of the object oriented programming feature that enables an object to have more than one meanings depending on the context. In other words, overloading refers to, the reuse of the same operator or the function name for more than one operations or functions.

1. Operator overloading can be applied only to the operators that are available. No new operator is created.
2. An overloaded operator is required to have a minimum of one user-defined operator.
3. Basic operation of an operator must not change.
4. Overloaded operators can't be overridden.
5. Binary arithmetic operators like +, *, /, % can be overloaded
6. When a member function or a friend function overloads the binary operators, it takes one or two explicit arguments appropriately.
7. When a member function or a friend function overloads the unary operators it takes no two explicit argument or one reference argument appropriately.
8. The precedence and the number of arguments of an operator doesn't change the result of operator overloading.

Q45. What is operator overloading? Write a syntax to overload operators.

Ans :

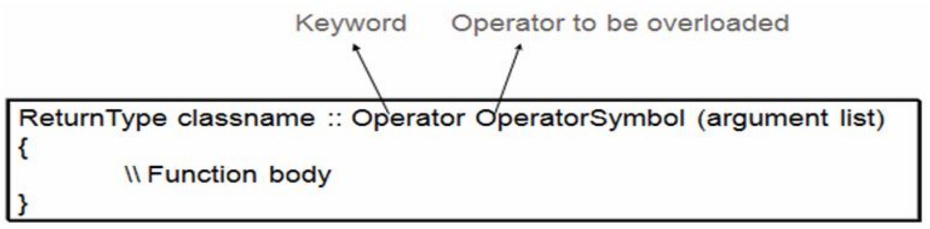
Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.



Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded.Operator that are not overloaded are follows

- ➢ Class member access operators (., .*)
- ➢ Scope resolution operator (::)
- ➢ Size operator (sizeof)
- ➢ Conditional operator (?:)

Syntax



```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
        \\ Function body
}
```

Q46. Explain the rules to be followed in operator overloading.

Ans :

Rules to be Followed in Operator Overloading

1. Operator overloading can be applied only to the operators that are available. No new operator is created.
2. An overloaded operator is required to have a minimum of one user-defined operator.
3. Basic operation of an operator must not change.
4. Overloaded operators can't be overridden.
5. Binary arithmetic operators like +, *, /, % can be overloaded
6. When a member function or a friend function overloads the binary operators, it takes one or two explicit arguments appropriately.
7. When a member function or a friend function overloads the unary operators it takes no two explicit argument or one reference argument appropriately.
8. The precedence and the number of arguments of an operator doesn't change the result of operator overloading.

Q47. How to overload operators in C++ programming? explain.

Ans :

To overload an operator, a special operator function is defined inside the class as:

```
class className
{
........
public
     returnType operator symbol (arguments)
{
........
}
........
};
```

- Here, returnType is the return type of the function.
- The returnType of the function is followed by operator keyword.
- Symbol is the operator symbol you want to overload. Like: +, <, -, ++
- You can pass arguments to the operator function in similar way as functions.

Q48. What is unary operator overloading? Write a program to implement unary operator overloading.

Ans :

The unary operators operate on a single operand and following are the examples of Unary operators:

- The increment (++) and decrement (–) operators.
- The unary minus (–) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj–.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

```
// demonstration of unary operator overloading
#include <iostream>
using namespace std;
class Distance
{
  private:
    int feet;              // 0 to infinite
    int inches;            // 0 to 12
  public:
                           // required constructors
    Distance(){
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i){
      feet = f;
      inches = i;
    }
    // method to display distance
    void displayDistance()
    {
      cout << "F: " << feet << " I:" << inches
<<endl;
    }
    // overloaded minus (-) operator
    Distance operator- ()
    {
      feet = -feet;
      inches = -inches;
      return Distance(feet, inches);
    }
};
int main()
{
   Distance D1(11, 10), D2(-5, 11);
   -D1;                          // apply negation
   D1.displayDistance();         // display D1
   -D2;                          // apply negation
   D2.displayDistance();         // display D2
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

OUTPUT

```
F: -11 I:-10
F: 5 I:-11
```

Q49. What is binary operator over loading? Write a c++ program to overlaod binary operator.

Ans :

```
#include<iostream>
#include<conio.h>
//Standard namespace declaration
using namespace std;
class overloading
{
 int value;
 public:
 void setValue(int temp)
 {
     value = temp;
 }
 overloading operator+(overloading ob)
 {
  overloading t;
  t.value=value+ob.value;
  return(t);
 }
void display()
{
 cout<<value<<endl;
}
};
//Main Functions
int main()
{
   overloading obj1,obj2,result;
   int a,b;
   cout<<"Enter the value of Complex Numbers a,b:";
   cin>>a>>b;
   obj1.setValue(a);
   obj2.setValue(b);
   result = obj1+obj2;
   cout<<"Input Values:\n";
   obj1.display();
   obj2.display();
   cout<<"Result:";
   result.display();
   getch();
   return 0;
}
```

Sample Output

```
Enter the value of Complex Numbers a,b:10
5
Input Values:
10
5
Result:15
```

Q50. Explain how to overload I/O operators with an example program.

Ans :

Overloading I/O operator

- Overloaded to perform input/output for user defined datatypes.
- Left Operand will be of types ostream& and istream&
- Function overloading this operator must be a Non-Member function because left operand is not an Object of the class.
- It must be a friend function to access private data members.

You have seen above that **<<** operator is overloaded with **ostream** class object cout to print primitive type value output to the screen. Similarly you can overload **<<** operator in your class to print user-defined type to screen. For example we will overload **<<** in **time** class to display time object using cout.

```
time t1(3,15,48);
cout << t1;
```

*****NOTE:** When the operator does not modify its operands, the best way to overload the operator is via friend function.

```
// demonstration overloading '<<' Operator to print time object
#include< iostream.h>
#include< conio.h>
class time
{
 int hr,min,sec;
 public:
  time()
  {
   hr=0, min=0; sec=0;
  }
   time(int h,int m, int s)
  {
   hr=h, min=m; sec=s;
  }
friend ostream& operator << (ostream &out, time &tm);  //overloading '<<' operator
};
ostream& operator<< (ostream &out, time &tm)          //operator function
{
 out << "Time is " << tm.hr << "hour : " << tm.min << "min : " << tm.sec << "sec";
 return out; }
void main()
{
 time tm(3,15,45);
 cout << tm; }
```

Output

```
Time is 3 hour : 15 min : 45 sec.
```

Q51. Write a program to Overload Relational operator.

Ans :

You can also overload Relational operator like ==, !=, >=, <= etc. to compare two user-defined object.

```
// demonstration of overloading relational operator
        class time
        {
        int hr,min,sec;
         public:
          time()
          {
           hr=0, min=0; sec=0;
          }

          time(int h,int m, int s)
          {
           hr=h, min=m; sec=s;
          }
        friend bool operator==(time &t1, time &t2);     // overloading '==' operator
        };
        bool operator== (time &t1, time &t2)             // operator function
        {
        return ( t1.hr == t2.hr &&
               t1.min == t2.min &&
               t1.sec == t2.sec );
        }
```

Short Question and Answers

1. Benefits of Object Oriented Programming.

Ans :

Benefits (or) Merits

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple objects to coexist without any interference.
- It is possible to map objects in the problem domain to those objects in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in an implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects make the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.
- Polymorphism can be implemented i.e. behavior of functions or operators or objects can be changed depending upon the operations.

2. Applications of OOP Technology.

Ans :

OOP has become one of the programming buzzwords today. There appears to be a great deal excitement and interest among software engineers in using OOP. Applications of OOP are beginning to gain importance in many areas. The most popular application of object- oriented programming, up to now, has been in the area of user interface design such as windows. Hundreds of windowing systems have been developed, using the OOP techniques.

Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of applications because it can simplify a complex problem. The promising areas for application of OOP include:

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

3. What is a Class ?

Ans :

A class is an abstract data type that groups the data and its associated functions. It can also hide the data and functions if required.

A class specification consists of two parts,

(i) Declaration of a class and

(ii) Definitions of member functions of a class.

The members scope and types are described by the class declaration. And, the implementation of member functions are described by the class function definitions.

General Form

```
class classname
{
access_specifier 1 :mem_var1;
⋮
mem_func1;
⋮
access_specifier2 :mem_var2;
⋮
mem_func2;
}
```

The class keyword indicates an abstract data type called name of a class and body of a class includes member variables and member function declarations.

4. How classes provide data encapsulation.

Ans :

A class is an abstract data type that groups the data and its associated functions. It can also hide the data and functions if required.

A class specification consists of two parts.

(i) Declaration of a class and

(ii) Definitions of member functions of a class.

The members scope and types are described by the class declaration. And, the implementation of member functions are described by the class function definitions.

Encapsulation is a mechanism of binding data members and corresponding methods into a single module classed class, inorder to protect them from being accessed by the outside code. An instance of a class can be called as an object and it is used to access the members of a class. In encapsulation, objects are treated as 'block boxes' since each object performs specific task.

The data and functions available in a class are called as members of a class. The data defined in the class are called as member variables or data members and the functions defined are called as member functions.

The main idea behind the concept of encapsulation is to obtain high maintenance and to handle the application's code.

5. What are Member Functions ?

Ans :

Member functions are the functions, which have their declaration inside the class definition and works on the data members of the class. The definition of member functions can be inside or outside the definition of class.

If the member function is defined inside the class definition it can be defined directly, but if its defined outside the class, then we have to use the scope resolution :: operator along with class name along with function name.

6. What is the need for private members?

Ans :

The classes contain both the functions and variables that are to be used in the program. Sometimes the class members might need to be used only within the class internally. It means the statements which are outside the class should not have access to these members. With this, the critical data can be protected from being modified.

The solution for this is to make the members private. This can be done by preceding the class members with private keyword. When a variable is declared as private, the values can be only stored in it through public member function. This function is the only means for the application to access the private variables.

7. What are constructors ?

Ans :

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors iitialize values to object members after storage is allocated to the object.

Syntax:

```
class A
{
 int x;
 public:
 A();          //Constructor
};
```

While defining a constructor you must remember that the name of constructor will be same as the name of the class, and constructors never have return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

Example of constructor

```
class A
{
 int i;
 public:
 A(); //Constructor declared
};
A::A()   // Constructor definition
{
 i=1;
}
```

8. Explain the types of constructors.

Ans :

(i) Default Constructor

Default constructor is the constructor which doesn't take any argument. It has no parameter.

(ii) Parameterised Constructors

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

(iii) COPY Constructor

Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type. It is usually of the form X (X &), where X is the class name.he compiler provides a default Copy Constructor to all the classes.

9. What is Constructor Overloading?

Ans :

Just like other member functions, constructors can also be overloaded. Infact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other with parameter.

You can have any number of Constructors in a class that differ in parameter list.

// demonstration of overloading constructor

```
class Student
{
 int rollno;
 string name;
 public:
 Student(int x)
 {
  rollno=x;
  name="None";
 }
 Student(int x, string str)
 {
  rollno=x ;
  name=str ;
 }
};
int main()
{
 Student A(10);
 Student B(11,"Ram");
}
```

In above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

In the above case if we write Student S; in **main()**, it will lead to a compile time error, because we haven't defined default constructor, and compiler will not provide its default constructor because we have defined other parameterized constructors.

10. What are private member functions?

Ans :

Usually member data are made private while functions (or methods) are made public. There might be instances where you might want to make certain functions private (i.e. you may not want the user to directly access these functions). Private functions can only be called from within public member functions. These functions are also called 'helper functions' Why do we need them?

Let's take the example of the class 'batsman'. After every match the user will enter the batsman's new score and then he will have to call two functions to update the batsman's record (i.e. the user has to call update_best ( ) and update_worst ( )). It is unnecessary to bother the user with this kind of a double function call. Why should the user access these two functions directly? Instead of this, we could define another function called update ( ) which will call update_best ( ) and update_worst ( ). In this way the user needs to only call one function after every match.

The idea of classes is to restrict user access. We don't want the user to access data or functions unnecessarily. So, we will make update_best ( ) and update_worst ( ) as private functions while update ( ) will be a public function.

11. Friend Functions

Ans :

Friend functions are actually not class member function. Friend functions are made to give private access to non-class functions. You can declare a global function as friend, or a member function of other class as friend.

Hence, friend functions can access private data members by creating object of the class. Similarly we can also make function of other class as friend, or we can also make an entire class as friend class.

When we make a class as friend, all its member functions automatically become friend functions.

Friend Functions is a reason, why C + + is not called as a pure Object Oriented language. Because it violates the concept of Encapsulation.

12. properties of friend function.

Ans :

1. A friend function is not a member function of a class to which it is declared as friend. Since, this friend function is out of scope of that particular class. Therefore, the objects of that class can't be used to call this friend function. These functions are invoked directly.
2. It can be declared either in public or private section of a class.
3. It can be invoked with out any object.
4. It is used in operator overloading.
5. It is called like any other normal function.
6. It can be called without including the dot(.) operator.
7. It takes objects of class type as arguments.
8. It is not allowed to access private members of a class. These private members are accessed only through dot and scope resolution operators.

13. Define overloading.

Ans :

Overloading is one of the object oriented programming feature that enables an object to have more than one meanings depending on the context. In other words, overloading refers to, the reuse of the same operator or the function name for more than one operations or functions.

1. Operator overloading can be applied only to the operators that are available. No new operator is created.
2. An overloaded operator is required to have a minimum of one user-defined operator.
3. Basic operation of an operator must not change.

4. Overloaded operators can't be overridden.
5. Binary arithmetic operators like +, *, /, % can be overloaded
6. When a member function or a friend function overloads the binary operators, it takes one or two explicit arguments appropriately.
7. When a member function or a friend function overloads the unary operators it takes no two explicit argument or one reference argument appropriately.
8. The precedence and the number of arguments of an operator doesn't change the result of operator overloading.

14. Explain the rules to be followed in operator overloading.

Ans :

1. Operator overloading can be applied only to the operators that are available. No new operator is created.
2. An overloaded operator is required to have a minimum of one user-defined operator.
3. Basic operation of an operator must not change.
4. Overloaded operators can't be overridden.
5. Binary arithmetic operators like +, *, /, % can be overloaded
6. When a member function or a friend function overloads the binary operators, it takes one or two explicit arguments appropriately.
7. When a member function or a friend function overloads the unary operators it takes no two explicit argument or one reference argument appropriately.
8. The precedence and the number of arguments of an operator doesn't change the result of operator overloading.

Choose the Correct Answer

1. What is default visibility mode for members of classes in C++ ? [a]
   (a) Private (b) Public
   (c) Protected (d) Depends
2. Which of the following keywords are used to control access to a class member ? [a]
   (a) protected (b) switch
   (c) goto (d) for
3. How we can define member function outside the class ? [d]
   (a) Using union (b) Using structure
   (c) Using pointers (d) Using scope resolution
4. Data members and member functions are enclosed within ? [c]
   (a) union (b) structure
   (c) class (d) array
5. Which among following is correct way of declaring object of a class ? [a]
   (a) ClassnameObjectname; (b) Class ClassnameObjectname;
   (c) Class Classname Object Objectname; (d) Classname Object Objectname;
6. How we can access data members using objects ? [d]
   (a) object@datamember (b) object*datamember
   (c) object->datamember (d) object.datamember
7. What is actual syntax of destructor in c++ ? [d]
   (a) !Classname( ) (b) @Classname( )
   (c) $Classname( ) (d) ~Classname( )
8. Which operators can not be overloaded ? [b]
   (a) Binary operator (b) Ternary operator
   (c) Unary operator (d) All can be overloaded
9. Which constructor does not initialise any data member [a]
   (a) dummy (b) default
   (c) copy (d) Parameterised
10. How many destructors can a class have ? [b]
   (a) 0 (b) 1
   (c) 2 (d) n

Fill in the blanks

1. ________ provides a template the describes the structure and behaviour of an object.
2. Data hiding is provided through ________ visibility label.
3. Variables of a classes are called ________.
4. An object is ________ an of class.
5. ________ function breaks the rules of data hiding.
6. ________ is the first member function to be executed when the object of that class is created.
7. If new operator is used, then the constructor is known as ________.
8. Name of the destructor is preceded by the ________ symbol.
9. Operators can be redefined using ________ concept.
10. ________ and ________ operators are already overloaded in C++.

ANSWERS

1. Class
2. Private
3. Objects
4. Instance
5. Friend
6. Constructor
7. Dynamic constructor.
8. ~
9. Operator overloading
10. = and &

UNIT III

Inheritance: Introduction, Protected Members and Class Access, Base Class Access Specification, Constructors and Destructors in Base and Derived Classes, Redefining Base Class Functions, Polymorphism and Virtual Member Functions, Abstract Base Classes and Pure Virtual Functions, Multiple Inheritance. C++ Streams: Stream Classes, Unformatted I/O Operations, Formatted I/O Operations.

3.1 INHERITANCE

3.1.1 Introduction

Q1. What is Inheritance and explain it with syntax and benefits.

Ans :

Reusability is yet another important .feature of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again. It would not only save time and money but also reduce frustration and increase reliability. For instance, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

Fortunately, C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called inheritance (or derivation). The old class is referred to as the base class and the new one is called the derived class or subclass.

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base classes is called single inheritance and one with several base classes is called multiple inheritance. On the other hand, the traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance. The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance. The direction of arrow indicates the direction of inheritance.

The process of obtaining the data members and methods from one class to another class is known as inheritance. It is one of the fundamental features of object-oriented programming.

Important points

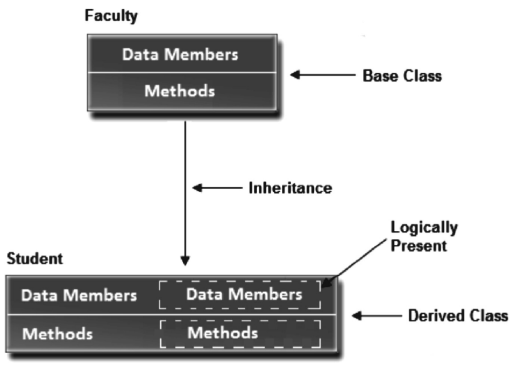
- In the inheritance the class which is give data members and methods is known as base or super or parent class.
- The class which is taking the data members and methods is known as sub or derived or child class.

Syntax

```
class subclass_name : superclass_name
{
        // data members
        // methods
}
```

Real life example of inheritance

The real life example of inheritance is child and parents, all the properties of father are inherited by his son



In the above diagram data members and methods are represented in broken line are inherited from faculty class and they are visible in student class logically.

Benefits of inheritance

If we develop any application using this concept than that application have following advantages,

- Application development time is less.
- Application take less memory.
- Application execution time is less.
- Application performance is enhance (improved).
- Redundancy (repetition) of the code is reduced or minimized so that we get consistence results and less storage cost.

Q2. What are base and derived classes? Explain them with an example.

Ans :

Base & Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

Consider a base class Shape and its derived class Rectangle as follows:

```
// demonstration of base and derived classes
#include<iostream>
usingnamespace std;
// Base class
{
    public:
    void setWidth(int w)
{
    width = w;
}
    void setHeight(int h)
{
    height = h;  }
    protected:
    int width;
    int height;  };
    // Derived class
    classRectangle:publicShape
{
    public:
    int getArea()
{
    return(width * height);
}       };
    int main(void)
{
    RectangleRect;
    Rect.setWidth(5);
    Rect.setHeight(7);
    // Print the area of the object.
    cout <<"Total area: "<<Rect.getArea()<<
    endl;
    return0;
}
```

When the above code is compiled and executed, it produces the following result:

Output

```
Total area:35
```

Q3. Explain briefly about various types of Inheritance

Ans :

Tyes of Inheritance

Based on number of ways inheriting the feature of base class into derived class it have five types they are:

i) Single inheritance

ii) Multi-level inheritance

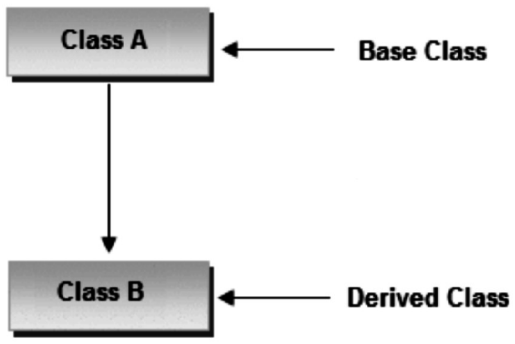
iii) Hierarchical inheritance

iv) Multiple inheritance

v) Hybrid inheritance

i) Single Inheritance

In single inheritance there exists single base class and single derived class.



ii) Multi level inheritances

In multi level inheritance there exists single base class, single derived class and multiple intermediate base classes.

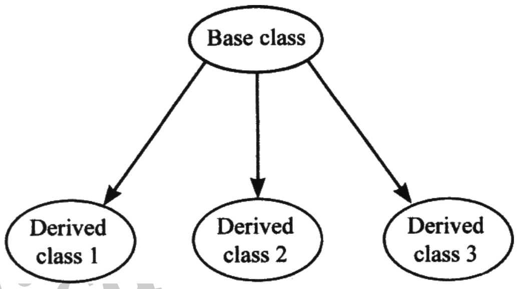
Single base class + single derived class + multiple intermediate base classes.

iii) Hierarchical inheritance

Hierarchical inheritance is the process of deriving many classes from a single base class. All the derived classes can be further inherited by some other classes in the same way.

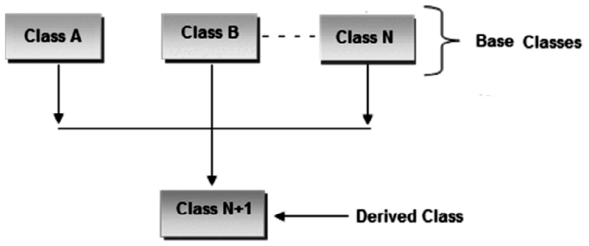
Syntax

```
class A
{
   //Body of class A
};
class B: visibility A
{
      //Body of class B
};
class C: visibility A
{
      //Body of class C
}
class D: visibility B
{
      //Body of class D
};
```



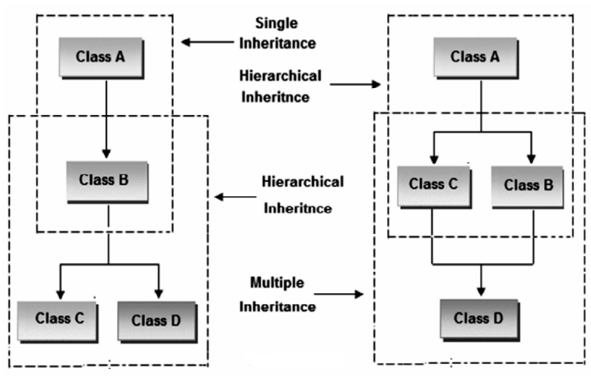
iv) Multiple inheritance

In multiple inheritance there exist multiple classes and singel derived class.



v) Hybrid inheritance

Combination of any inheritance type



Rahul Publications

Q4. Describe briefly about single inheritance. With an example.

Ans :

Single Inheritance

Single inheritance is the process of deriving a class from a single base class. It has only one base and one derived class. Only a single derived class can inherit the properties and behavior from the single base class.

Shows a base class B and a derived class D. The class B contains one private data member, one public data member, and three public member functions. The class D contains one private data member and two public member functions.

Example

```
linclude <iostream>
using namespace std;
class B
{
    int a;        // private; not inheritable
  public:
    int b;    // public; ready for inheritance
    void get_ab();
    int get_a(void);
    void show_a(void);
};
class D : public B      // public derivation
{
    int c;
  public:
    void mul(void);
    void display(void);
};
//................................................
void B :: get_ab(void)
{
      a = 5; b = 10;
}
int B :: get_a()
{
    return a;
}
void B :: show_a()
{
    cout << "a = " « a « "\n" •
}
void D :: mul()
{
    c = b * get a();
}
void D :: display()
{
    cout <<  "a = " <<a << "\n";
    cout <<  "b = "   << b << "in";
    cout <<  "c = "   << c << "\n\n";
}
//.............................................................

int main()
{
    D d;
    d.get_ab();
    d.mul();
    d.show_a();
    d.display();
    d.b = 20;
    d.mul();
    d.display();
    return 0;
}
```

Output

```
a = 5
a = 5
b = 10
c = 150

a = 5
b = 20
c = 100
```

Q5. Describe briefly about Multi-level inheritance with an example.

Ans :

It is not uncommon that a class is derived from another derived class as shown in Fig. The class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. The class B is known as intermediate base class since it provides a link for the inheritance between A and C. The chain ABC is known as inheritance path.

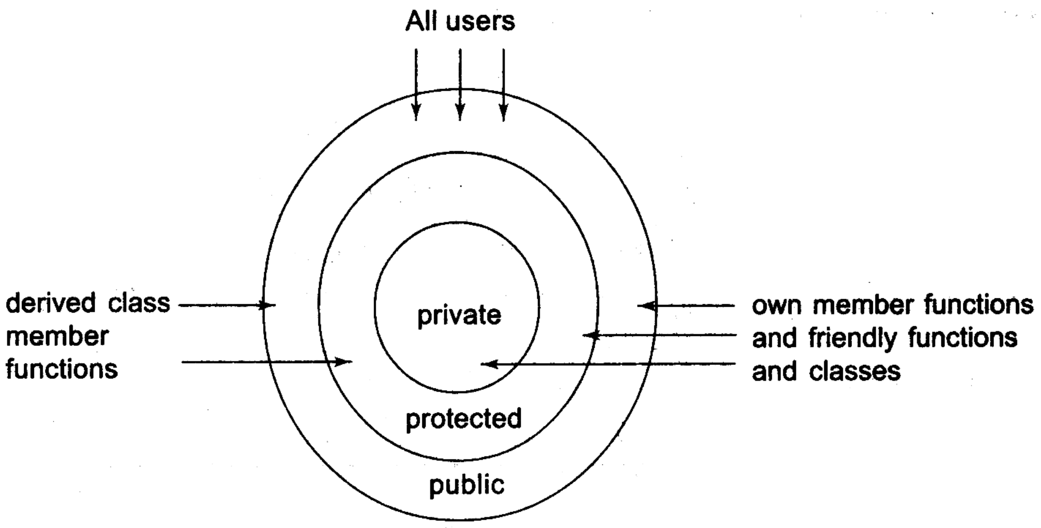

All users
derived class member functions
private
protected
public
own member functions and friendly functions and classes

Fig. A simple view of access control to the members of a class

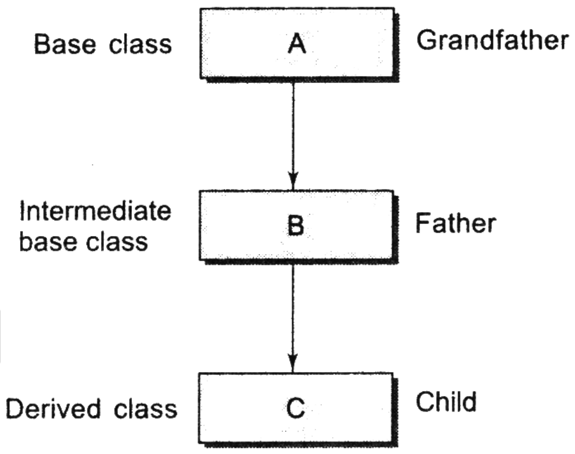


Fig. Multilevel inheritance

Example

```
#inc1ude <iostream>
using namespace std
class student
{
    protected:
            int roll_number;
    public:
            void get_number(int);
            void put_number(void);
};
```

```
void student :: get_number(int a)
{
            roll number = a;
}
void student :: put number()
{
            cout <<"Roll Number: " << roll number <<"\n";
}
class test : public student                              //  First level derivation
{
protected:
   float sub1;
   float sub2;
public:
      void get_marks(float, float);
      void put_marks(void);
};
void test :: get_marks(float x, float y)
{
   sub1 = x;
   sub2 = y;
}
void test :: put_marks()
{
            cout <<"Marks in SUB1 = " << subl << "\n";
            cout << "Marks in SUB2 = " << sub2 << "\n";
class result : public test                               // Second level  derivation
{
      float total;                                       // private by default
   public:
      void display(void);
};
void result :: display(void)
{
            total = sub1 + sub2;
            put_number();
            put_marks();
            cout << "Total = " << total << "\n";
}
int main()
```

```
{
        result student1;                    // student1 created
        student1.get_number(111);
        student1.get_marks(75.0, 59.5);
        student1.display( );
    return 0;
}
```

Example

```
Roll Number: 111
Marks in SUB1 = 75
Marks in SUB2 = 59.5
Total = 134.5
```

3.1.2 Multiple Inheritance

Q6. Describe briefly about multiple inheritance with an example.

Ans :

Multiple Inheritance

A class can inherit the attributes of two or more classes as shown in Fig. This is known as multiple inheritance. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and the intelligence of another.

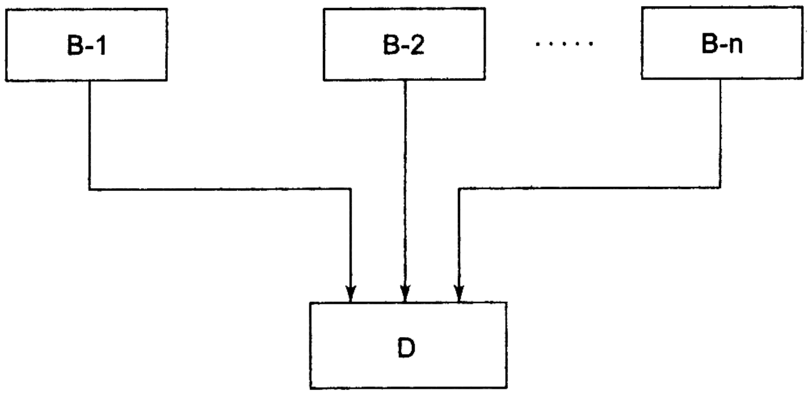


Fig.: Multiple inheritance

The syntax of a derived class with multiple base classes is as follows

```
class D: visibility B-1, visibility B-2 ...
{
    .....
    ..... (Body of D)
    .....
};
```

Multiple Inheritance

```
linclude <iostream>
using namespace std;
class M
{
   protected:
           int m;
   public:
           void get_m(int);
};
class N
{
   protected:
      int n;
      public:
           void get_n(int);
};
   class P : public M, public N
{
   public:
      void display(void);
};
void M :: get_m(int x)
{
        m = x;
}
void N :: get_n(int y)
{
        n = y;
}
      void P :: display(void)
{
      cout << "m = " << m << "\n";
      cout << "n = " << n << "\n";
      cout << "m*n = " << m*n << "\n"
}
int main()
{
           P p;
           p.get_m(10)
           p.get_n(20)
           p.display( )
           return 0;
}
```

Output

```
      m = 10
      n = 20
      m*n = 200
```

Q7. Describe briefly about Hierachical Inheritance with an example.

Ans :

Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level.

As an example, Figure shows a hierarchical classification of students in a university Another example could be the classification of accounts in a commercial bank as shown in Fig. All the students have certain things in common and, similarly, all the accounts possess certain common features.

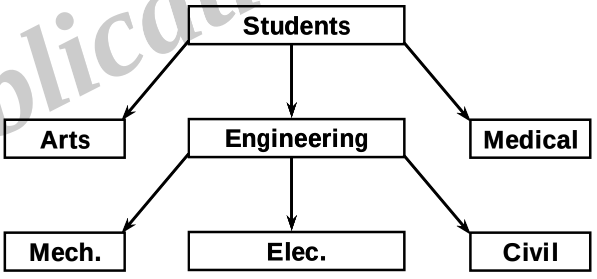


Fig.: Hierachical classification of students

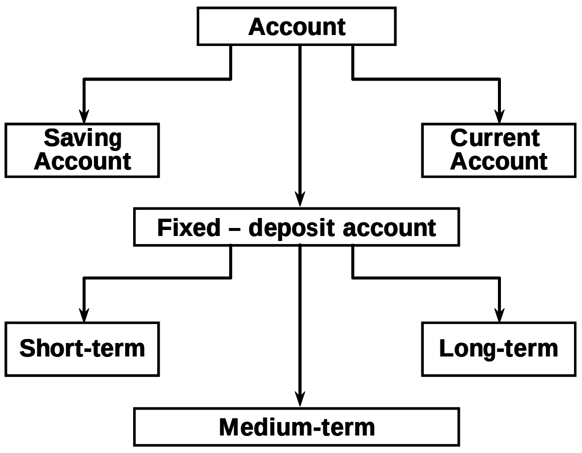


Fig.: Classification of bank account

In C++, such problems can be easily converted into class hierarchies. The base class will include all the features that are common to the subclasses. A subclass can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes and so on.

Q8. Describe briefly about Hybrid inheritance with an example..

Ans :

It is a situations where we need to apply two or more types of inheritance to design a program. Assume that we have to give weightage for sports before finalising the results. The weightage for sports is stored in a separate class called sports. The new inheritance relationship between the various classes would be as shown in Fig.

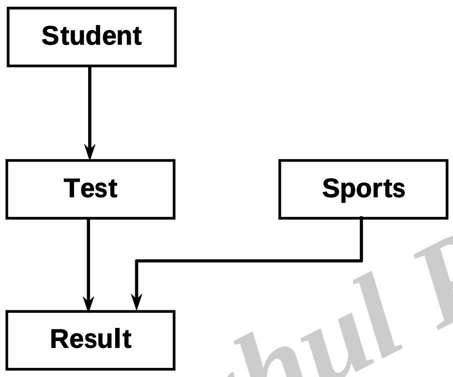


Fig.: Multilevel, multiple inheritance

The sports class might look like:

```
class sports
{
   protected:
      float score;
   public:
      void get_score(float);
      void put_score(void);
};
```

The result will have both the multilevel and multiple inheritances and its declaration would be as follows:

```
class result : public test, public sports
{
        .....
        .....
};
```

Where test itself is a derived class from student. That is

```
class test : public student
{
        .....
        .....
};
```

Example

```
#include <iostream>
using namespace std;
class student
{
   protected:
      int roll_number;
   public:
      void get_number(int a)
      {
           roll_number = a;
      }
        void put_number(void)
        {
        cout << "Roll No: " << roll_number
               << "\n'';
        }
};
class test : public student
{
   protected:
        float part1, part2;
   public:
        void get_marks(float x, float y)
        {
                  parti = x; part2 = y;
        }
        void put_marks(void)
        {
        cout  << "Marks obtained:" << "\n"
           << "Parti = " << part1 <<"\n"
           << "Part2 = " << part2 << "\n";
        }
};
class sports
{
   protected:
         float score;
   public:
         void get_score(float s)
         {
               score = s;
         }
         void put_score(void)
         {
               cout << "Sports wt: " << score << "\n\n
         }
};
class result : public test, public sports
{
      float total;
public:
      void display(void);
};
void result :: display(void)
{
         total = part1 + part2 + score;
         put_number();
         put_marks();
         put_score();
         cout << "Total Score: " << total << "\n";
}
int main()
{
      result student_1;
      student_1.get_number(1234);
      student_1.get_marks(27.5, 33.0);
      student_1.get_score(6.0);
      student_1.display();
      return 0;
}
```

Output

```
Roll No: 1234
Marks obtained:
Parti = 27.5 Part2 = 33
Sports wt: 6
Total Score: 66.5
```

3.2 Protected Members and Class Access

Q9. Define protected member and how to access it?

Ans :

The protected access specifier

When dealing with inherited classes, things get a bit more complex.

C++ has a third access specifier that we have yet to talk about because it's only useful in an inheritance context. The protected access specifier allows the class the member belongs to, friends, and derived classes to access the member. However, protected members are not accessible from outside the class.

```
class ProtectedAccess
{
        protected:          // protected access specifier
        int x;              // Data Member Declaration
        void display();     // Member Function decaration
}
```

Protected data members, can be accessed directly using dot (.) operator inside the subclass of the current class, for non-subclass we will have to follow the steps same as to access private data member.

```
class Base
{
        public:
        int m_public; // can be accessed by anybody
        private:
        int m_private; // can only be accessed by Base members and friends (but not derived classes)
protected:
        int m_protected; // can be accessed by Base members, friends, and derived classes
};
class Derived: public Base
{
public:
Derived()
{
        m_public = 1; // allowed: can access public base members from derived class
        m_private = 2; // not allowed: can not access private base members from derived class
        m_protected = 3; // allowed: can access protected base members from derived class
}
};
int main()
{
```

```
Base base;
        base.m_public = 1; // allowed: can access public      //members from outside class
        base.m_private = 2; // not allowed: can not access  //private members from outside class
        base.m_protected = 3; // not allowed: can not       //access protected members from outside class
}
```

In the above example, you can see that the protected base member m_protected is directly accessible by the derived class, but not by the public.

Q10. Write a program to demonstrate how to access protected member from base class.

Ans :

```
#include<iostream>
usingnamespace std;
classBox{
protected:
double width;
};
classSmallBox:Box                    // SmallBox is the derived class. {
public:
void setSmallWidth(double wid );
double getSmallWidth(void);
};
// Member functions of child class
doubleSmallBox::getSmallWidth(void)
{
return width ;
}
voidSmallBox::setSmallWidth(double wid )
{
width = wid;
}
          // Main function for the program
int main(){
SmallBox box;
          // set box width using member function
box.setSmallWidth(5.0);
cout <<"Width of box : "<< box.getSmallWidth()<< endl;
return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Width of box : 5
```

3.3 Base Class Access Specification

Q11. What are Different kinds of baseclass, and their impact on access? Explain briefly with an example.

Ans :

First, there are three different ways for classes to inherit from other classes: public, private, and protected.

To do so, simply specify which type of access you want when choosing the class to inherit from:

```
// Inherit from Base publicly
class Pub: public Base
{
};
// Inherit from Base privately
class Pri: private Base
{
};
// Inherit from Base protectedly
class Pro: protected Base
{
};
class Def: Base // Defaults to private inheritance
{
};
```

If you do not choose an inheritance type, C++ defaults to private inheritance

That gives us 9 combinations: 3 member access specifiers (public, private, and protected), and 3 inheritance types (public, private, and protected).

i) Public inheritance

Public inheritance is by far the most commonly used type of inheritance. In fact, very rarely will you see or use the other types of inheritance, so your primary focus should be on understanding this section. Fortunately, public inheritance is also the easiest to understand. When you inherit a base class publicly, inherited public members stay public, and inherited protected members stay protected. Inherited private members, which were inaccessible because they were private in the base class, stay inaccessible.

Access specifier in base class	Access specifier when inherited publicly
Public	Public
Private	Inaccessible
Protected	Protected

Here’s an example showing how things work:

```
class Base
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};
class Pub: public Base // note: public inheritance
{
    // Public inheritance means:
    // Public inherited members stay public (so m_public is treated as public)
    // Protected inherited members stay protected (so m_protected is treated as protected)
    // Private inherited members stay inaccessible (so m_private is inaccessible)
public:
    Pub()
    {
        m_public = 1; // okay: m_public was inherited as public
        m_private = 2; // not okay: m_private is inaccessible from derived class
        m_protected = 3; // okay: m_protected was inherited as protected
    }
};
int main()
{
    // Outside access uses the access specifiers of the class being accessed.
    Base base;
    base.m_public = 1; // okay: m_public is public in Base
    base.m_private = 2; // not okay: m_private is private in Base
    base.m_protected = 3; // not okay: m_protected is protected in Base
    Pub pub;
    pub.m_public = 1; // okay: m_public is public in Pub
    pub.m_private = 2; // not okay: m_private is inaccessible in Pub
    pub.m_protected = 3; // not okay: m_protected is protected in Pub
```

This is the same as the example above where we introduced the protected access specifier, except that we've instantiated the derived class as well, just to show that with public inheritance, things work identically in the base and derived class.

Public inheritance is what you should be using unless you have a specific reason not to.

Rule: Use public inheritance unless you have a specific reason to do otherwise.

ii) Private inheritance

With private inheritance, all members from the base class are inherited as private. This means private members stay private, and protected and public members become private.

Note that this does not affect the way that the derived class accesses members inherited from its parent! It only affects the code trying to access those members through the derived class.

```
class Base
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};
class Pri: private Base          // note: private inheritance
{
                 // Private inheritance means:
                 // Public inherited members become private (so m_public is treated as private)
                 // Protected inherited members become private (so m_protected is treated as private)
                 // Private inherited members stay inaccessible (so m_private is inaccessible)
public:
    Pri()
    {
        m_public = 1;    // okay: m_public is now private in Pri
        m_private = 2;   // not okay: derived classes can't access private members in the base class
        m_protected = 3;   // okay: m_protected is now private in Pri
    }
};
int main()
{
                          // Outside access uses the access specifiers of the class being accessed.
                          // In this case, the access specifiers of base.
```

```
Base base;
base.m_public = 1;          // okay: m_public is public in Base
base.m_private = 2;         // not okay: m_private is private in Base
base.m_protected = 3;       // not okay: m_protected is protected in Base
Pri pri;
pri.m_public = 1;           // not okay: m_public is now private in Pri
pri.m_private = 2;          // not okay: m_private is inaccessible in Pri
pri.m_protected = 3;        // not okay: m_protected is now private in Pri
```

To summarize in table form:

Access specifier in base class	Access specifier when inherited publicly
Public	Private
Private	Inaccessible
Protected	Private

Private inheritance can be useful when the derived class has no obvious relationship to the base class, but uses the base class for implementation internally. In such a case, we probably don't want the public interface of the base class to be exposed through objects of the derived class (as it would be if we inherited publicly). Private inheritance also ensures the derived class must use the public interface of the base class, ensuring encapsulation is upheld.

But in reality, this is rarely used.

iii) Protected inheritance

Protected inheritance is the last method of inheritance. It is almost never used, except in very particular cases. With protected inheritance, the public and protected members become protected, and private members stay inaccessible.

Because this form of inheritance is so rare, we'll skip the example and just summarize with a table:

Access specifier in base class	Access specifier when inherited protectedly
Public	Protected
Private	Inaccessible
Protected	Protected

Example

```
        class Base
{
public:
        int m_public;
private:
        int m_private;
protected:
        int m_protected;
};
```

Base can access its own members without restriction. The public can only access m_public. Derived classes can access m_public and m_protected.

```
class D2 : private Base // note: private inheritance
{
        // Private inheritance means:
        // Public inherited members become private
        // Protected inherited members become private
        // Private inherited members stay inaccessible
public:
        int m_public2;
private:
        int m_private2;
protected:
        int m_protected2;
};
```

D2 can access its own members without restriction. D2 can access Base's m_public and m_protected members, but not m_private. Because D2 inherited Base privately, m_public and m_protected are now considered private when accessed through D2. This means the public can not access these variables when using a D2 object, nor can any classes derived from D2.

```
class D3 : public D2
{
        // Public inheritance means:
        // Public inherited members stay public
        // Protected inherited members stay protected
        // Private inherited members stay inaccessible
public:
        int m_public3;
private:
        int m_private3;
protected:
        int m_protected3;
};
```

D3 can access its own members without restriction. D3 can access D2's m_public2 and m_protected2 members, but not m_private2. Because D3 inherited D2 publicly, m_public2 and m_protected2 keep their access specifiers when accessed through D3. D3 has no access to Base's m_private, which was already private in Base. Nor does it have access to Base's m_protected or m_public, both of which became private when D2 inherited them.

3.4 Constructors and Destructors in Base and Derived Classes

Q12. What is the use of constructors and destructors in Inheritance.

Ans :

Constructors, Destructors, in Inheritance

The constructors are used to initialize member variables of the object, and the destructor is used to destroy the object. The compiler automatically invokes constructors and destructors. The derived class does not require a constructor, if the base class contains a zero-argument constructor. In case the base class has a parameterized constructor, then it is essential for the derived class to have a constructor. The derived class constructor passes arguments to the base class constructor. In inheritance, normally derived classes are used to declare objects.

Hence, it is necessary to define constructors in the derived class. When an object of a derived class is declared, the constructors of the base and derived classes are executed.

Q13. What is the Order of Constructor Call in Inheritance

When a default or parameterized constructor of a derived class is called, the default constructor of a base class is called automatically. As you create an object of a derived class, first the default constructor of a base class is called after that constructor of a derived class is called.

To call parameterized constructor of a base class you need to call it explicitly as shown below.

```
        Student(string szName,int iYear,string szUniversity): Person(szName, iYear)
{
}
```

Below program will show the order of execution that the default constructor of base class finishes first after that the constructor of a derived class starts. For example, there are two classes with single inheritance:

```
//base class
classPerson
{
public:
        Person()
        {
                cout  <<"Default constructor of base class  called"<< endl;
        }
        Person(string lName,int year)
        {
                cout  <<"Parameterized constructor of  base class called"<< endl;
                lastName = lName;
                yearOfBirth = year;
        }
        string lastName;
        int yearOfBirth;
};
//derived class
classStudent:publicPerson
{
public:
        Student()
        {
               cout  < < "Default constructor of Derived  class called" < <  endl;
        }
               Student(string lName,int year,string univer)
        {
               cout  < < "Parameterized constructor of Derived class called" < <  endl;
               university  =  univer;
        }
        string university;
};
```

There is no explicit call of constructor of a base class. But on creating two objects of Student class using default and parameterized constructors, both times default constructor of a base class get called.

```
        Student  student1;
               // Using default constructor of Student class
        Student student2("John",1990,"London School of   Economics");
               // Calling parameterized constructor of
               // Student class
```

In both the above cases, default constructor of a base class is called before the constructor of a derived class.

```
        Default constructor of baseclass called
        Default constructor of Derivedclass called
        Default constructor of baseclass called
Parameterized constructor of Derivedclass called
```

When multiple inheritance is used, default constructors of base classes are called in the order as they are in inheritance list. For example, when a constructor of derived class is called:

```
        class derived:public class1,publicclass2
```

the order of constructors calls will be

class1 default constructor

class2 default constructor

derived constructor

If you want to call a parameterized constructor of the base class then this can be done using initializer list as shown below.

```
Student(string lName,int year,string univer): Person(lName, year)
{
cout < <"Parameterized constructor of Derived class works"< < endl;
university  = univer;
}
```

Above code means that you are calling parametrized constructor of the base class and passing two parameters to it. Now the output will be

```
Default constructor of baseclass works
Default constructor of Derivedclass works
Parameterized constructor of baseclass works
Parameterized constructor of Derivedclass works
```

Now you can see that parameterized constructor of the base class is called from derived class constructor.

Q14. How to implement Base class Default Constructor in Derived class Constructors. Show with an example.

Ans :

```
// demonstranttion of base and derived class            // constructors
class Base
{ int x;
public:
Base() { cout < < "Base default constructor"; }
};
class Derived : public Base
{ int y;
public:
Derived() { cout < < "Derived default constructor"; }
Derived(int i) { cout < < "Derived parameterized constructor"; }
};
int main()
{
Base b;
Derived d1;
Derived d2(10);
}
```

Q15. How to implement Base class Parameterized Constructor in Derived class Constructor. Show with an example program.

Ans :

We can explicitly mention to call the Base class's parameterized constructor when Derived class's parameterized constructor is called.

```
        // demonstration of parameterised
           constructors in
        // derived class
class Base
{ int x;
public:
Base(int i)
{ x = i;
cout < < "Base Parameterized Constructor";
}    };
class Derived : public Base
{ int y;
public:
Derived(int j) : Base(j)
{ y = j;
cout < < "Derived Parameterized Constructor";
}  };
int main()
{
Derived d(10) ;
cout < < d.x ;          // Output will be 10
cout < < d.y ;          // Output will be 10
}
```

Q16. Why is Base class Constructor called inside Derived class ?

Ans :

Constructors have a special job of initializing the object properly. A Derived class constructor has access only to its own class members, but a Derived class object also have inherited property of Base class, and only base class constructor can properly initialize base class members. Hence all the constructors are called, else object wouldn't be constructed properly.

Q17. What is Virtual base class? How to use it in program.

Ans :

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.

```
// demonstration of virtual base classes
class A
{
    public:
        int i;
};

class B : virtual public A
{
    public:
        int j;
};
class C: virtual public A
{
    public:
        int k;
};
class D: public B, public C
{
    public:
        int sum;
};
int main()
{
    D ob;
    ob.i = 10;   // unambiguous since only one copy of
                 // i is inherited.
    ob.j = 20;
    ob.k = 30;
    ob.sum = ob.i + ob.j + ob.k;
    cout << "Value of i is : "<< ob.i<<"\n";
    cout << "Value of j is : "<< ob.j<<"\n"; cout   << "Value of k is :"<< ob.k<<"\n";
    cout << "Sum is : "<< ob.sum <<"\n";
    return 0;
}
```

Q18. Define Object slicing.

Ans :

When a Derived Class object is assigned to Base class, the base class' contents in the derived object are copied to the base class leaving behind the derived class specific contents. This is referred as Object Slicing. That is, the base class object can access only the base class members. This also implies the separation of base class members from derived class members has happened.

Object slicing is a concept where additional attributes of a derived class object is sliced to form a base class object.

Object slicing doesn't occur when pointers or references to objects are passed as function arguments since both the pointers are of the same size.

Object slicing will be noticed when pass by value is done for a derived class object for a function accepting base class object.

Object slicing could be prevented by making the base class function pure virtual there by disallowing object creation.

Q19. Write a program to demonstrate object slicing.

Ans :

```
//Demonstrate the concept of object slicing
#include <iostream>
using namespace std;
class Base {
int data1;
int data2;
public:
Base(int a, int b) {
data1 = a;
data2 = b;
}
virtual void display() {
```

```
cout << "I am Base class" << endl;
}
};
class Derived : public Base {
int data3;
public:
Derived(int a, int b, int c) : Base(a, b) {
data3 = c;
}
void display() {
cout << "I am Derived class" << endl;
}
};
void somefunc ( Base obj )
{
obj.display();
}
int main()
{
Base b(10, 20);
Derived d(100, 200, 300);
somefunc(b);
somefunc(d);
}
```

OUTPUT

```
I am Base class
I am Base class
```

3.5 Redefining Base Class Functions

Q20. Define function redefining?

Ans :

A redefined function is a method in a descendant class that has a different definition than a non-virtual function in an ancestor class. Don't do this. Since the method is not virtual, the compiler chooses which function to call based upon the static type of the object reference rather than the actual type of the object.

For example, if you have an Animal *george , and george = new Monkey; where Monkey inherits from Animal, if you say george->dosomething() the Animal.dosomething() method is called, even though george is a Monkey (even if a Monkey.dosomething() method is available).

Q21. Write a program to demonstration function redefining.

Ans :

So I have a base class named Geometry, a child named Rectangle and a grandchild named Box. All Geometry holds is the name of the instance, Rectangle has Length and Width, and Box has Height. For my assignment I am supposed to call a function computeSurfaceArea() which returns the area of the shape. So for Geometry it should be 0, Rectangle is Length*Width and Box is 2 * (Length * Width + Length * Height + Width * Height). So I figured that the best way to do that would be a different compute Surface Area() at each level. But when I run the following main, only the function from Geometry is used:

```
#include"Box.h"
#include<iostream>
    using namespace std;
    void report(Geometry *temp);
    int main(){
        Geometry *A[3];
        A[0] = new Geometry("Geom 1");
        A[1] = new Rectangle("Rec 2",4,5);
        A[2] = new Box("Box 3", 2,3,4);
        report(A[0]);
        report(A[1]);
        report(A[2]);
        return 0;
}
```

```
void report(Geometry *temp){
cout << "–Geometry Report–" << endl;
cout<< "Type:" <<temp->getType() << endl;
cout << "Name: " << temp->getName() << endl;
      cout << "Surface: " << temp-> computeSurface() << endl;
      cout << "Volume: " << temp- >computeVolume() << endl << endl;
      return;
}
```

3.6 Polymorphism and Virtual Member Functions

Q22. What is polymorphism? Explain dif-ferent types of polymorphism with examples.

OR

What is a polymorphism? Explain different types of polymorphism with example program.

Ans :

Polymorphism is one of the important object oriented programming concepts. It is a mechanism through which one operation or a function can take many forms depending upon the type of objects. This is a single operator or function that can be used in many ways.

Consider an example for adding two variables,

$\boxed{\text{Sum} = x + y}$

Here, x and y can be integer numbers

Sum = 2 + 5

(or) float numbers

Sum = 2.5 + 7.5

The result of 'Sum' depends upon the values passed to it.

Types of Polymorphism

There are two types of polymorphism. They are,

(i) Compile time polymorphism

(ii) Runtime polymorphism.

(i) Compile Time Polymorphism

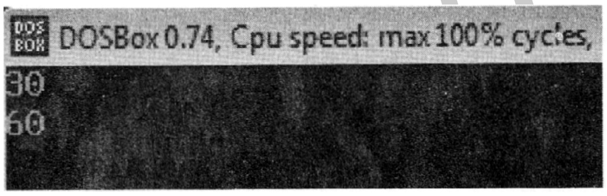
In compile time polymorphism, the most appropriate member function is called by comparing the type and the number of arguments by the compiler at compile time. It is also known as early binding or static linking or static binding.

Example

```
#include<iostream.h>
#include<conio.h>
class Add
```

```
{
        public:
        void sum(int a, int b)
        {
            cout< <  a+b;
        }
        void sum(int a, int b, int c)
        {
            cout< <a+b+c;
        }
};
void main()
{
            clrscr();
            Add obj;
            obj.sum(10, 20);
            cout«endl;
            obj.sum(10, 20, 30);
}
```

Output



```
DOSBox 0.74, Cpu speed: max 100% cycles,
30
60
```

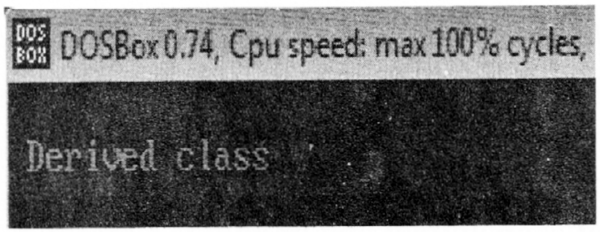
(ii) Runtime Polymorphism

In run time polymorphism, the most appropriate member function is called at runtime i.e., while the program is executing and the linking of function with a class occurs after compilation. Hence, it is called Tate binding'. It is also known as dynamic binding. It is implemented using virtual functions and the pointers to objects.

Example

```
#include<iostream.h>
#include<conio.h>
class Baseclass
{
public:
virtual void show()
{
cout< <"Base class \n";
}
};
class Derivedclass: public Baseclass
{
public: void show()
{
cout< <"\n Derived class \n";
}
};
int main(void)
{
clrscr();
Baseclass *bp = new Derivedclass;
bp->show(); // Run-time polymorphism
getch();
return 0;
}
```

Output



```
DOSBox 0.74, Cpu speed: max 100% cycles,

Derived class
```

Q23. Define virtual function.

Ans :

Virtual Function

Virtual is a keyword that is used to achieve polymorphism and resolve the ambiguity raised in multipath inheritance. An object can inherit the properties of a derived class object which intum inherits the properties of base class object. Ambiguity arises while calling the inherited objects. Such ambiguities are resolved using virtual functions.

A function is made virtual by placing virtual keyword before the function name. A virtual function when defined in the base class can also be redefined by all the derived classes. It provides one interface to have multiple forms. Several versions of virtual function are accessed using the appropriate class objects pointed by the base pointers.

Syntax

virtual retumtype functionname();

Q24. State the rules associated with virtual function.

Ans:

The rules associated with virtual function are as follows,

1. Virtual functions cannot be static.
2. Virtual function definitions must be available in the base class even if it is not used.
3. They must be the members of some class.
4. They can be a friend functions to some other classes.
5. Object pointers are used to access the virtual functions.
6. If the virtual function definition occurs in the base class then there is no need to redefine it in the derived class and when the invocation of such function occurs then it automatically calls the base class function.
7. An object of the base type cannot be accessed with a derived class pointer.
8. Incrementing or decrementing a base class pointer that points to the derived class will not result in pointing to the next derived class object.
9. The prototype of a virtual function in the base class must match with its derived class versions. Otherwise, they will be considered as overloaded functions which ignore the concept of virtual functions.

Q25. Write a C++ program for virtual function.

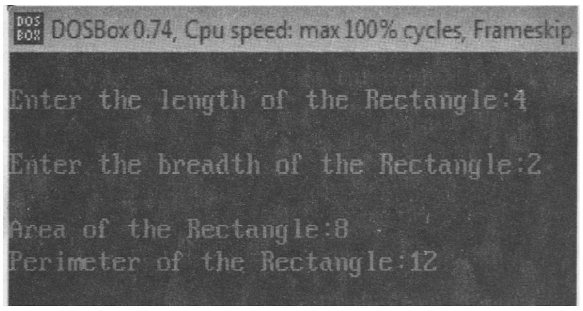
Ans :

Program

```
#include<iostream.h>
#include<conio.h>
class Rectangle
{
public:
   float len, br;
Rectangle()
{
}
Rectangle(float 1, float b)
{
len = 1;
br = b;
}
void get()
{
cout<<"\nEnter the length of the Rectangle:";
cin>>len;
cout<<"\nEnter the breadth of the Rectangle:";
cin>>br;
  }
virtual void compute()
{
float x = 2.0;
float y = len + br;
x = x*y;
cout<<"\nPerimeter of the Rectangle:"<<x;
}
};
```

```
class Area: public Rectangle
{
public:
void compute()
{
cout<<"\nArea of the Rectangle:"<<len*br;
}
};
void main()
{
clrscr();
Area a;
Rectangle *ptr;
ptr = &a;
ptr ->get();
ptr ->compute();
Rectangle r(ptr -> len, ptr -> br);
ptr = &r;
ptr ->compute();
getch();
}
```

Output



```
Enter the length of the Rectangle:4

Enter the breadth of the Rectangle:2

Area of the Rectangle:8
Perimeter of the Rectangle:12
```

3.7 Abstract Base Classes

26. Define Abstract Class? What are the characteristics of abstract class.

Ans :

Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

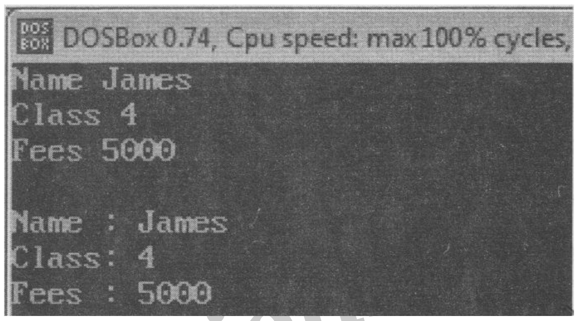
Characteristics of Abstract Class

- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Example

```
#include <iostream.h>
#include <conio.h>
const int max = 80;
class first
{
  protected:
    char name [max];
    char els [max] ;
  public:
    virtual void insert( )=0;
    virtual void show( )=0;
};
class second: public first
{
  protected:
    int fees;
  public:
    void insert()
  {
    cout<<"Name";
    cin>>name;
    cout<<"Class";
    cin>>cls;
    cout<<"Fees";
    cin>>fees;
}
void show( )
{
    cout<<"\nName:"<<name<<"\n";
            cout<<"Class:"<<cls<<"\n";
            cout<<"Fees:"<<fees<<"\n";
      }
};
void main( )
{
clrscr(); second si; sl.insert(); sl.show(); getchf);
}
```

Output



```
Name James
Class 4
Fees 5000

Name : James
Class: 4
Fees : 5000
```

3.8 Pure Virtual Functions

27. Define pure virtual function with syntax.

Ans :

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function. Pure virtual function doesn't have body or implementation. We must implement all pure virtual functions in derived class.

Pure virtual function is also known as abstract function.

A class with at least one pure virtual function or abstract function is called abstract class. We can't create an object of abstract class. Member functions of abstract class will be invoked by derived class object.

General Syntax of Pure Virtual Function takes the form:

1. class class_name //This denotes the base class of C++ virtual function
2. {
3. public:
4. virtualvoid virtualfunctioname()=0//This denotes the pure virtual function in C++
5. };

28. Write the Differentiate between virtual function and pure virtual function

Ans :

Basis For Comparison	Virtual Function	Pure Virtual Function
Basic	'Virtual function' has their definition in the base class.	'Pure Virtual Function' has no definition in the base class.
Declaration	virtual funct_name(parameter_list) {. . . . .};	virtual funct_name(parameter_list)=0;
Derived class	All derived classes may or may not override the virtual function of the base class.	All derived classes must override the virtual function of the base class.
Effect	Virtual functions are hierarchical in nature; it does not affect compilation if any derived classes do not override the virtual function of the base class.	If all derived classes fail to override the virtual function of the base class, the compilation error will occur.
Abstract class	No concept.	If a class contains at least one pure virtual function, then it is declared abstract.

29. Write a program to demonstrate pure virtual functions.

Ans :

```
 #include<iostream.h>
 #include<conio.h>
class BaseClass      //Abstract class
{
      public:
        virtual void Display1()=0;//Pure virtual function or abstract function
        virtual void Display2()=0;//Pure virtual function or abstract function
       void Display3()
       {
             cout<<"\n\tThis is Display3() method of Base Class";
       }
};
class DerivedClass : public BaseClass
{
      public:
       void Display1()
       {
        cout< < “\n\tThis is Display1() method
                  of Derived Class”;
    }
    void Display2()
    {
      cout< < “\n\tThis is Display2() method
                     of Derived Class”;
    }
};
void main()
{
    DerivedClass D;
    D.Display1();
      // This will invoke Display1() method of
      // Derived Class
    D.Display2();
      // This will invoke Display2() method of
      // Derived Class
    D.Display3();
      // This will invoke Display3() method of
      // Base Class
}
```

Output :

```
This is Display1() method of Derived Class
This is Display2() method of Derived Class
This is Display3() method of Base Class
```

3.9 C++ Streams

3.9.1 Stream Classes

Q30. Describe the hierarchy of stream classes in C++.

Ans :

Stream Classes

Stream classes are set of classes, whose functionality depends on console file operations. They are declared in header file “iostream.h”. It is mandatory for a programmer to include this header file, whenever a program is written using the functions supported by these stream classes.

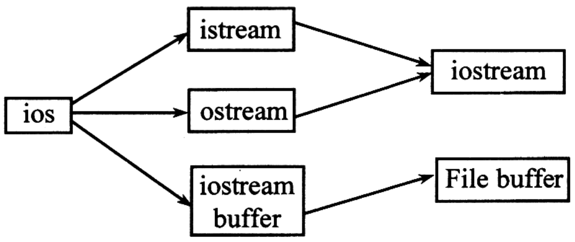


Fig.(a): Stream Classes

Figure (a) represents the hierarchy of stream classes.

From the above hierarchical structure it can be inferred that,

(i) ios is the parent class

(ii) istream, ostream are child classes

(iii) iostreambuffer is a member variable object of ios

(iv) The iostream class is a child class of both istream class and ostream class.

The other streams include classes istream_ withassign, ostream_withassign and iostream _ withassign. They are used to append the required assignment operators.

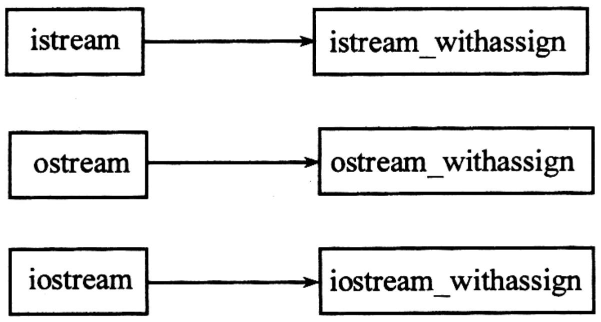


Fig. (b): Other Stream Classes

Types of Stream Classes

The different stream classes include,

(a) ios

(b) istream

(c) ostream

(d) iostream

(e) istream jwithassign

(f) ostreamwithassign

(g) iostream_withassign.

(a) ios

ios is an input and output stream class, that performs both formatted and unformatted I/O operations. This class basically is a pointer that points to a buffer iostreambuffer. Moreover, the information related to the state of iostream buffer is maintained by ios stream class.

(b) istream

istream is a derived class of ios stream class which is used to manage both formatted data and unformatted data that is available in streambuf object. In addition to this, istream provides input of formatted data properties of istream.

Properties of istream

(i) The istream class overloads the extraction operator (>>).

(ii) It declares functions like peek( ), tellg( ), seekg( ), getline( ), read( ).

(c) ostream

ostream is an output stream class derived from ios class. This class handles formatting of output data and is used to provide general purpose output.

Properties of ostream

(i) The ostream class overloads the insertion operator(<<).

(ii) It declares functions like tellp( ), put( ), write( ), seek( ).

(d) iostream

iostream is the derived class of istream and ostream and therefore supports all the functions of its base classes. It is an input and output stream that is used to manage both input and output operations.

(e) istream_withassign

istream withassign is a stream class derived from istream class and is used while providing input using cin object.

(f) ostream_withassign

ostream_withassign is a stream class derived from ostream class and is used while generating output using cout.

(g) iostream_withassign

iostream_withassign is a combination of both istream_withassign and ostream withassign and it can be called as bidirectional stream.

3.9.2 Unformatted I/O Operations

Q31. Explain in detail various non-formatted (or) unformatted I/O functions.

Ans :

Non-formatted or Unformatted Input/Output Functions

Non-formatted or unformatted Input/output functions are the simple and basic I/O functions of C++. They are the means of the data transfer between the memory and the file in binary form. They can operate only on the data of type 'char'.

Different functions in this category are as follows,

(i) getchar( )

(ii) putchar( )

(iii) gets( )

(iv) puts( )

(v) getch( )

(vi) putch( ).

(i) getchar( )

This function returns single character entered from keyboard. No arguments are required for this function. By calling this function, user can read a string.

Syntax: var = getchar( );

Here, var is an identifier of char type.

(ii) putchar(var): This function displays a single character on an output device.

Syntax: putchar(var);

(iii) gets( ): This function reads an input string.
Syntax: gets(var);

Here, var is a character array

(iv) **puts(var):** This function displays string stored in var on keyboard.

Syntax: puts(var);

(v) **getch( ):** getch(**)** is an unformatted I/O function defined in 'conio.h' header file. It is an input function that takes single character as input and does not display (echo) it on screen.

Syntax: int getch(void)

or

variableName = getch();

(vi) **putch( ):** putch**()** is an unformatted I/O function defined in 'conio.h' header file. This function is used for displaying a single alphanumeric character to the screen.

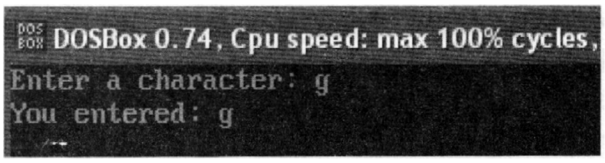
Syntax: putch(variable)

Programs

Example on getchar() and putchar()

```
#include<iostream.h>
#include<conio.h> void main()
{
    char ch; clrscr();
    cout<<"Enter a character:";
    ch = getchar();
    cout<<"You entered:";
    putchar(ch); getch();
    return 0;
}
```

Output

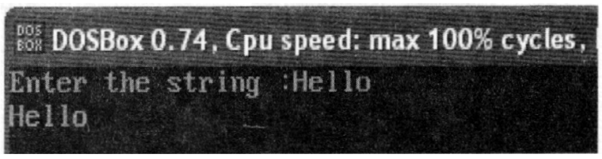


```
DOSBox 0.74, Cpu speed: max 100% cycles,
Enter a character: g
You entered: g
```

Example on gets( ) and puts( )

```
#include<iostream.h>
void main( )
{
    char a[25]; clrscr( );
    cout<<"Enter the string :
    gets(a);
    puts(a);
    getch();
    return;
}
```

Output

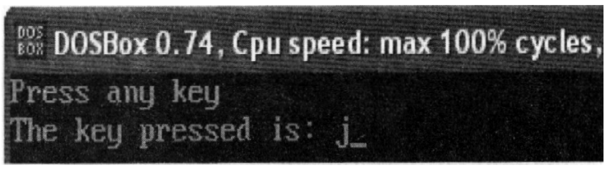


```
DOSBox 0.74, Cpu speed: max 100% cycles,
Enter the string :Hello
Hello
```

Example on getch( ) and putch( )

```
#inc 1 ude<io stream. h>
#include<conio.h>
int main()
{
    char ch;
    cout<<"Press any key\n"
    ch = getch();
    cout<<"The key pressed is:";
    putch(ch);
    return 0;
}
```

Output



```
DOSBox 0.74, Cpu speed: max 100% cycles,
Press any key
The key pressed is: j_
```

3.9.3 Formatted I/O Operations

Q32. Explain formatted I/O functions with examples.

Ans :

Formatted I/O Operations

Formatted console I/O functions used in C++ for formatting the output are as follows,

1. ios class functions and flags
2. Manipulators
3. Custom/user-defined manipulators.

1. ios Class Functions and Flags

ios Class Function

The various ios class functions are as follows,

(i) width( )

(ii) precision( )

(iii) fill( )

(iv) self( )

(i) ios::width( )

A function width( ) is a number function that is used to set the width of a field in order to display the output value. The declaration of this function can be done in either of the following ways,

(a) int width( );

This function on its invocation returns the present settings of the width.

(b) int width(int);

This functions on its invocation sets the width size as integer value (which is specified within the argument) and returns the previous settings of the width.

However, it should be assured that the width size for each and every item is specified separately.

(ii) ios::precision( )

The precision( ) function is also a number function that specifies the number of digits that are to be displayed after the decimal point where floating point numbers are to be printed. The declaration of this function can be done in either of the following ways.

(a) int precision( );

This function on its invocation returns the present setting of floating point precision.

(b) int precision(int);

This function on its invocation sets the floating point and returns the previous setting of this precision.

(iii) ios::fill( )

The function fill( ) is used to fill the empty locations by other required characters. The declaration of this function can also be done in either of the following ways,

(a) char fill( );

This function on its invocation returns the present settings of fill character.

(b) char fill(char);

This function on its invocation resets the fill character and finally returns the previous fill setting.

(iv) ios::self( )

self( ) is another number function of ios char which sets the formatting flag when invoked. The declaration of this function can be done in either of the following ways,

(a) DataType self (argl, arg2);

This function removes the bits marked in var as defined by the data number x and then resets the bits marked in var 'x'.

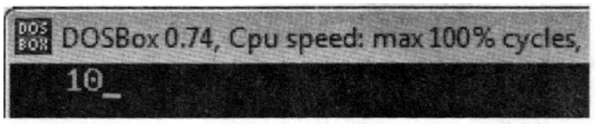
(b) DataType self (datatype)

This function on its invocation sets the flag in accordance to the bits marked in the parameterized data type.

Example

```
#include<iostream.h>
#include<conio.h> int main()
{
    clrscr();
    cout. width) 10);
    int a = cout.width(5);
    cout<<a;
    return 0;
}
```

Output


DOSBox 0.74, Cpu speed: max 100% cycles,

```
10_
```

This program has two width function calls. The first width( ) function call sets the column width at position 10. The next width( ) function call sets the column width at position 5 and returns the first column position i.e., 10. This value is taken by “a”. Finally, cout function displays 10 at column position 5.

Flags with Bitfields

The various bit-fields along with their format flags are,

(i) ios::adjustfield

(ii) ios::floatfield

(iii) ios::basefield.

(i) ios::adjustfield

This bit-field is a data member associated with the setf() function. It specifies the action of the value required by the output. The action of the output value in bit-field ios::adjustfield is as follows,

ios::left (Left justified output)

ios::right (Right justified output)

ios:internal (Padding after sign and base).

The declaration of ios::adjustfield can be done in the following manner, static const long adjustfield;

(ii) ios::floatfield

This bit-field is another data member associated with a setfi) function. It sets the floating point notation to scientific notation or fixed point notation.

The different ios “float field along with its flag format are as follows,

ios: :scientific(scientific notation)

ios::fixed (fixed point notation)

The declaration of ios::floatfield can be done in the following manner,

static const long floatfield;

(iii) ios::basefield

This field is also a data member associated with setf() function. It sets the notations to decimal base, octal base and hexadecimal base.

The different ios::basefield along with their flag format are as follows,

ios::dec (Decimal base)

ios: :oct (Octal base)

ios::hex (hexadecimal base).

The declaration of ios::basefield can be done in the following manner,

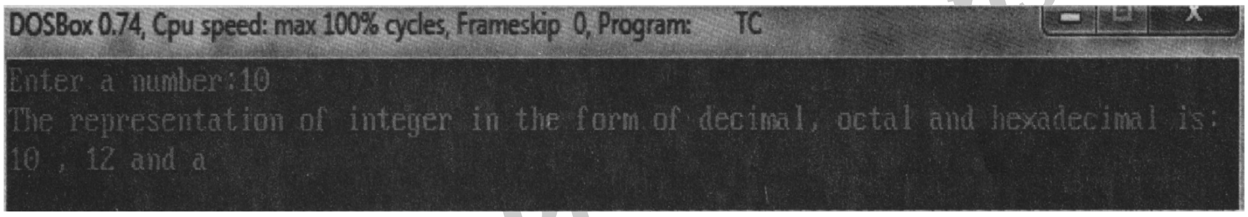
static const long basefield;

Example

```
#include<iostream.h>
#include<conio.h> void main()
{
```

```
int number;
clrscr();
cout<<"Enter a number:";
cin>>number;
cout<<"The representation of integer in the form of decimal, octal and hexadecimal is:";
cout.setfiios::dec, ios::basefield);
cout<<number«",";
cout.setf(ios: :oct, ios: ibasefield);
cout<<number<<" and
cout. setf(ios ::hex, ios::basefield);
cout<<number;
getch( );
}
```

Output


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
Enter a number:10
The representation of integer in the form of decimal, octal and hexadecimal is:
10 , 12 and a
```

Flags without Bitfields

The different flags that does not have corresponding bit fields are as follows,

(i) **ios ::showbase:** This flag makes use of base indicator to display an output.

(ii) **ios ::showpos:** This flag displays the preceding positive number.

(iii) **ios ::showpoint:** This flag displays the trailing decimal point and zeros.

(iv) **ios ::uppercase:** This flag makes use of capital letters to display the hexadecimal (hex) output.

(v) **ios ::skipws:** This flag skips the white spaces that appear in the input data.

(vi) **ios :: unitbuf:** This flag flushes away the streams after completing all the insertions

(vii) **ios ::stdio:** This flag sets the stream in accordance to the standard input and output of C++.

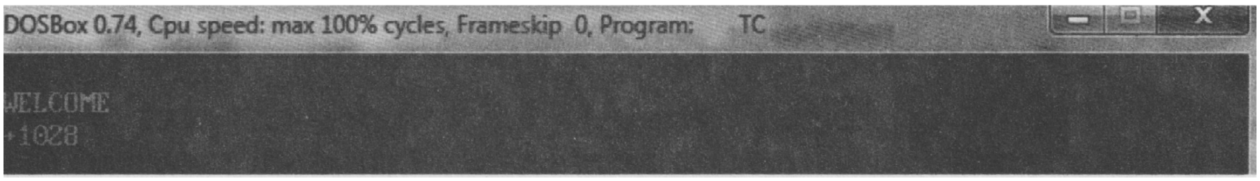
(viii) **ios ::boolalpha:** This flag converts the boolean values in the form of text i.e., either "true" or "false".

Example

```
#include<iostream.h>
#include<conio.h> int main()
{
    clrscr( );
    cout.setf(ios::skipws);
    cout«endl<<"WELCOOME";
    cout.setf(ios::showpos);
```

```
        cout<<endl<<1028;
        getch();
        return 0;
    }
```

Output


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
WELCOME
+1028
```

In this program, the setting ios::boolalpha converts the values to text i.e., 1 is converted to true. Second, ios:: skipws removes the white space and displays WELCOME. Finally, ios::showpos shows positive sign(+) before the number i.e., +1028.

2. Manipulators

Manipulators are used for manipulating (controlling) the output formats. They are similar to that of ios class member functions and flags. The only difference is that, the ios class member function returns the previous settings, whereas, the manipulator does not return the previous setting.

Manipulator along with cout statement can be written as,

```
cout<<manip 1<<manip2<<var 1;
```

Where,

manipl, manip2 are manipulators

varl is C++ variable.

Pre-defined Manipulators

The various pre-defined manipulators are as follows,

(a) **setw(int x):** This manipulator sets the field width to the fixed size of x.

(b) **setbase:** This manipulator sets the base of the number system.

(c) **setprecision (int** y): This manipulator sets the floating point precision toy.

(d) **setfill(char z):** This manipulator sets the fill character to the variable 'z' In other words, it sets the fill character to the character stored in variable 'z'.

(e) **setiosflagsflong d):** This manipulator sets the format flag to the variable 'd'.

(f) **resetiosflags (long d):** This manipulator eliminates the flags indicated by the variable 'd\

(g) **endl:** This manipulator divides (splits) into a new line and flushes the buffer stream.

(h) **skipws:** This manipulator skips (or removes) the blank spaces from the input data.

(i) **noskipws:** This manipulator does not skip blank spaces (white spaces) of the input data.

(j) **ends:** This manipulator closes the output string by adding null character to it.

(k) **flush:** This manipulator flushes off the buffer streams.

(l) **lock:** This manipulator grants lock on the file associated with the respective file handling function.

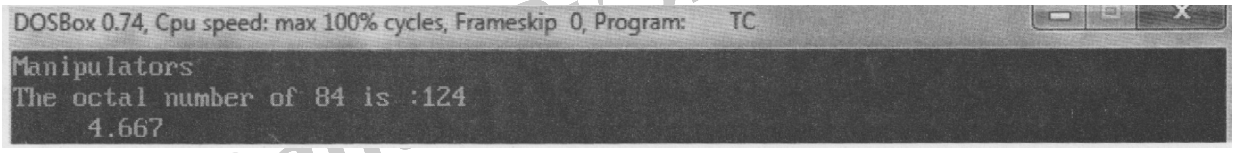
(m) **ws:** This manipulator eliminates the blank spaces present before the available first field.

(n) **hex, oct, dec:** This manipulator displays the hexadecimal, octal and decimal format for the number system.

Example

```
/*program to display formatted output using manipulators*/
#include<iostream.h>
#include<iomanip.h>
#include<conio.h> int main()
{
    clrscr( );
    cout<<setw(5)<<"Manipulators";
    cout<<setiosflags(ios::oct);
    cout<<"\n" << "The octal number of 84 is <<84;
    cout«endl;
    cout«setw( 10)<<setprecision(3)<<4.6666;
    getch();
    return 0;
    }
```

Output



```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program:    TC
Manipulators
The octal number of 84 is :124
     4.667
```

In this program, the manipulator setw(5) is used to set the width of field to '5'. So, the string 'Manipulator'is displayed at column 5. The 'setiosflag' manipulator is used to set the octal setting. So, the equivalent octal number 124 is displayed using cout statement. Finally, the manipulator setprecision (3) sets the number to '3 ' decimal point. So, the number 4.6666 is displayed as 4.667 at 10th column.

3. Custom/User-defined Manipulators

User-defined manipulators or custom-defined manipulators are nothing but the manipulators defined by the programmer or user in accordance to the requirement of the program.

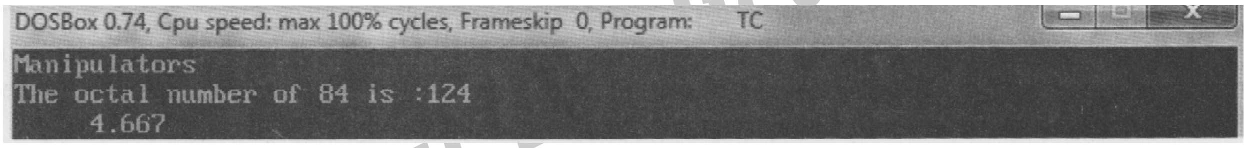
Syntax

```
ostream and manip name(ostream &output)
{
    statement 1;
    statement2;
    statement3;
    .
    .
    .
    return output;
}
```

Example

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
ostream &newline(ostream &output)
{
      output<<"\n";
      return output;
}
void main()
{
      clrscr();
      cout<<1<<newline<<2 << newline<<3;
      getch();
}
```

Output



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
Manipulators
The octal number of 84 is :124
     4.667
```

Short Question & Answers

1. What is Inheritance.

Ans :

Reusability is yet another important .feature of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again. It would not only save time and money but also reduce frustration and increase reliability. For instance, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

Fortunately, C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called inheritance (or derivation). The old class is referred to as the base class and the new one is called the derived class or subclass.

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base classes is called single inheritance and one with several base classes is called multiple inheritance. On the other hand, the traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance. The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance. The direction of arrow indicates the direction of inheritance.

The process of obtaining the data members and methods from one class to another class is known as inheritance. It is one of the fundamental features of object-oriented programming.

2. Various types of Inheritance

Ans :

Tyes of Inheritance

Based on number of ways inheriting the feature of base class into derived class it have five types they are:

i) Single inheritance

ii) Multi-level inheritance

iii) Hierarchical inheritance

iv) Multiple inheritance

v) Hybrid inheritance

3. Public inheritance

Ans :

Public inheritance is by far the most commonly used type of inheritance. In fact, very rarely will you see or use the other types of inheritance, so your primary focus should be on understanding this section. Fortunately, public inheritance is also the easiest to understand. When you inherit a base class publicly, inherited public members stay public, and inherited protected members stay protected. Inherited private members, which were inaccessible because they were private in the base class, stay inaccessible.

4. Private inheritance

Ans :

With private inheritance, all members from the base class are inherited as private. This means private members stay private, and protected and public members become private.

Note that this does not affect the way that the derived class accesses members inherited from its parent! It only affects the code trying to access those members through the derived class.

5. Protected inheritance

Ans :

Protected inheritance is the last method of inheritance. It is almost never used, except in very particular cases. With protected inheritance, the public and protected members become protected, and private members stay inaccessible.

Because this form of inheritance is so rare, we'll skip the example and just summarize with a table:

Access specifier in base class	Access specifier when inherited protectedly
Public	Protected
Private	Inaccessible
Protected	Protected

6. What is the use of constructors and destructors in Inheritance.

Ans :

The constructors are used to initialize member variables of the object, and the destructor is used to destroy the object. The compiler automatically invokes constructors and destructors. The derived class does not require a constructor, if the base class contains a zero-argument constructor. In case the base class has a parameterized constructor, then it is essential for the derived class to have a constructor. The derived class constructor passes arguments to the base class constructor. In inheritance, normally derived classes are used to declare objects.

Hence, it is necessary to define constructors in the derived class. When an object of a derived class is declared, the constructors of the base and derived classes are executed.

7. Define Object slicing.

Ans :

When a Derived Class object is assigned to Base class, the base class' contents in the derived object are copied to the base class leaving behind the derived class specific contents. This is referred as Object Slicing. That is, the base class object can access only the base class members. This also implies the separation of base class members from derived class members has happened.

- Object slicing is a concept where additional attributes of a derived class object is sliced to form a base class object.
- Object slicing doesn't occur when pointers or references to objects are passed as function arguments since both the pointers are of the same size.
- Object slicing will be noticed when pass by value is done for a derived class object for a function accepting base class object.
- Object slicing could be prevented by making the base class function pure virtual there by disallowing object creation.

8. Define function redefining?

Ans :

A redefined function is a method in a descendant class that has a different definition than a non-virtual function in an ancestor class. Don't do this. Since the method is not virtual, the compiler chooses which function to call based upon the static type of the object reference rather than the actual type of the object.

For example, if you have an Animal *george , and george = new Monkey; where Monkey inherits from Animal, if you say george->dosomething() the Animal.dosomething() method is called, even though george is a Monkey (even if a Monkey.dosomething() method is available).

9. What is polymorphism?

Ans :

Polymorphism is one of the important object oriented programming concepts. It is a mechanism through which one operation or a function can take many forms depending upon the type of objects. This is a single operator or function that can be used in many ways.

Consider an example for adding two variables,

$\boxed{\text{Sum} = x + y}$

Here, x and y can be integer numbers

Sum = 2 + 5

(or) float numbers

Sum = 2.5 + 7.5

The result of 'Sum' depends upon the values passed to it.

10. Types of Polymorphism

Ans :

There are two types of polymorphism. They are,

(i) Compile time polymorphism

(ii) Runtime polymorphism.

(i) Compile Time Polymorphism

In compile time polymorphism, the most appropriate member function is called by comparing the type and the number of arguments by the compiler at compile time. It is also known as early binding or static linking or static binding.

ii) Runtime Polymorphism

In run time polymorphism, the most appropriate member function is called at runtime i.e., while the program is executing and the linking of function with a class occurs after compilation. Hence, it is called Tate binding'. It is also known as dynamic binding. It is implemented using virtual functions and the pointers to objects.

11. Define virtual function.

Ans :

Virtual is a keyword that is used to achieve polymorphism and resolve the ambiguity raised in multipath inheritance. An object can inherit the properties of a derived class object which intum inherits the properties of base class object. Ambiguity arises while calling the inherited objects. Such ambiguities are resolved using virtual functions.

A function is made virtual by placing virtual keyword before the function name. A virtual function when defined in the base class can also be redefined by all the derived classes. It provides one interface to have multiple forms. Several versions of virtual function are accessed using the appropriate class objects pointed by the base pointers.

12. Rules associated with virtual function.

Ans:

The rules associated with virtual function are as follows,

1. Virtual functions cannot be static.
2. Virtual function definitions must be available in the base class even if it is not used.
3. They must be the members of some class.
4. They can be a friend functions to some other classes.
5. Object pointers are used to access the virtual functions.
6. If the virtual function definition occurs in the base class then there is no need to redefine it in the derived class and when the invocation of such function occurs then it automatically calls the base class function.
7. An object of the base type cannot be accessed with a derived class pointer.
8. Incrementing or decrementing a base class pointer that points to the derived class will not result in pointing to the next derived class object.

13. Define Abstract Class?

Ans :

Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.

Abstract class can have normal functions and variables along with a pure virtual function.

Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.

Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

14. Define pure virtual function.

Ans :

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function. Pure virtual function doesn't have body or implementation. We must implement all pure virtual functions in derived class.

Pure virtual function is also known as abstract function.

A class with at least one pure virtual function or abstract function is called abstract class. We can't create an object of abstract class. Member functions of abstract class will be invoked by derived class object.

15. Unformatted Input / Output Functions

Ans :

Non-formatted or unformatted Input/output functions are the simple and basic I/O functions of C++. They are the means of the data transfer between the memory and the file in binary form. They can operate only on the data of type 'char'.

Different functions in this category are as follows,

(i) getchar( )

(ii) putchar( )

(iii) gets( )

(iv) puts( )

(v) getch( )

(vi) putch( ).

(i) getchar( )

This function returns single character entered from keyboard. No arguments are required for this function. By calling this function, user can read a string.

Syntax: var = getchar( );

Here, var is an identifier of char type.

(ii) **putchar(var):** This function displays a single character on an output device.

Syntax: putchar(var);

(iii) **gets( ):** This function reads an input string. **Syntax:** gets(var);

Here, var is a character array

(iv) **puts(var):** This function displays string stored in var on keyboard.

Syntax: puts(var);

(v) **getch( ):** getch(**)** is an unformatted I/O function defined in 'conio.h' header file. It is an input function that takes single character as input and does not display (echo) it on screen.

Syntax: int getch(void)

or

variableName = getch();

(vi) **putch( ):** putch**()** is an unformatted I/O function defined in 'conio.h' header file. This function is used for displaying a single alphanumeric character to the screen.

Syntax: putch(variable).

16. Formatted I/O functions with examples.

Ans :

Formatted console I/O functions used in C++ for formatting the output are as follows,

1. ios class functions and flags
2. Manipulators
3. Custom/user-defined manipulators.

1. ios Class Functions and Flags

ios Class Function

The various ios class functions are as follows,

(i) width( )

(ii) precision( )

(iii) fill( )

(iv) self( )

2. Manipulators

Manipulators are used for manipulating (controlling) the output formats. They are similar to that of ios class member functions and flags. The only difference is that, the ios class member function returns the previous settings, whereas, the manipulator does not return the previous setting.

Manipulator along with cout statement can be written as,

cout<<manip 1<<manip2<<var 1;

Where,

manipl, manip2 are manipulators

varl is C++ variable.

3. Custom/User-defined Manipulators

User-defined manipulators or custom-defined manipulators are nothing but the manipulators defined by the programmer or user in accordance to the requirement of the program.

Syntax

```
ostream and manip name(ostream &output)
{
        statement 1;
        statement2;
        statement3;
        .
        .
        .
        return output;
}
```

17. When should we use the protected access specifier?

Ans :

With a protected attribute in a base class, derived classes can access that member directly. This means that if you change anything about that protected attribute (the type, what the value means, etc…), you'll probably need to change both the base class AND all of the derived classes.

Therefore, using the protected access specifier is most useful when you (or your team) are going to be the ones deriving from your own classes, and the number of derived classes is reasonable. That way, if you make a change to the implementation of the base class, and updates to the derived classes are necessary as a result, you can make the updates yourself (and have it not take forever, since the number of derived classes is limited).

Making your members private gives you better encapsulation and insulates derived classes from changes to the base class. But there's also a cost to build a public or protected interface to support all the access methods or capabilities that the public and/or derived classes need. That's additional work that's probably not worth it, unless you expect someone else to be the one deriving from your class, or you have a huge number of derived classes, where the cost of updating them all would be expensive.

Choose the Correct Answer

1. The default visibility mode while inheriting is ? [c]
   (a) public (b) protected
   (c) private (d) may be any of above
2. The process of deriving a class from another derived class is known as ? [d]
   (a) single inheritance (b) dual inheritance
   (c) multiple inheritance (d) multilevel inheritance
3. When a derived class inherits from many base classes, this process is known as ? [a]
   (a) multiple inheritance (b) multilevel inheritance
   (c) default inheritance (d) multiplex inheritance
4. Only one copy of the class is inherited, when it is defined as ? [a]
   (a) virtual (b) public
   (c) static (d) private
5. Object slicing can be resolved using ? [a]
   (a) pointers (b) References
   (c) Abstract Classes (d) All of these
6. < < operator is ? [b]
   (a) stream extraction operator (b) stream insertion operator
   (c) left shift operator (d) right shift operator
7. Which operator is used for input stream? [b]
   (a) > (b) > >
   (c) < < (d) > > >
8. Which of the following is the correct class of the object cout? [c]
   (a) iostream (b) Isteam
   (c) Ostream (d) Ifstream
9. Pick out the correct objects about the instantiation of output stream. [d]
   (a) cout (b) cerr
   (c) clog (d) All of the mentioned
10. Run time polymorphism can be achieved with______ . [c]
    (a) Virtual Base class (b) Container class
    (c) Virtual function (d) Both a and c

Fill in the blanks

1. The process by which objects of one class acquire the attributes of another class is known as__________?
2. The major goal of inheritance in C++ is __________.
3. If class A used the features of Class B , then A is called the __________ class and B is called the __________ class.
4. The technique of creating a new class from an existing class is called __________.
5. Inheritance is frequently used to implement __________ relation ship.
6. Ambiguity problem can be solved by using __________ keyword.
7. A function is declared virtual in the __________ class.
8. __________ is a member function that is declared within a base class and redefined by derived class.
9. __________ will be used with physical devices to interact from C++ program?
10. __________ must be specified when we construct an object of class ostream.

ANSWERS

1. Inheritance
2. To facilitate the reusability of code
3. base, derived
4. Inheritance
5. Is – a
6. virtual
7. Base
8. virtual function
9. streams
10. streambuf

UNIT IV

Exceptions: Introduction, Throwing an Exception, Handling an Exception, Object-Oriented Exception Handling with Classes, Multiple Exceptions, Extracting Data from the Exception Class, Re-throwing an Exception.

Templates: Function Templates–Introduction, Function Templates with Multiple Type, Overloading with Function Templates, Class Templates – Introduction, Defining Objects of the Class Template, Class Templates and Inheritance.

4.1 Exceptions

4.1.1 Introduction

Q1. What is an Exceptions? Explain various types of exceptions.

Ans :

Exceptions are the run-time errors that occurs during the program execution.

Exception occurs due to,

(i) Division-by-zero condition

(ii) Exceeding the bounds of an array

(iii) Running out of memory

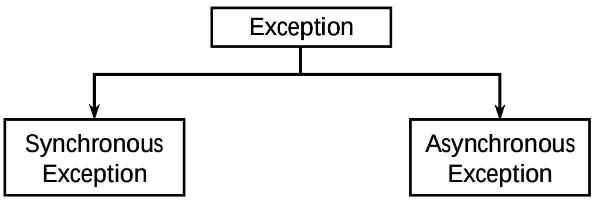
(iv) Falling short of memory

(v) Object initialization to an impossible value.

These exceptions can be handled in a systematic way by using three keywords try, throw and catch.

Types of Exceptions

The various types of exceptions are as follows,



Errors like out of range array index and overflow are the types of synchronous exceptions. The errors encountered due to the occurrence of the events that are not under the control of the program are called asynchronous exceptions.

Q2. List out various built-in exceptions.

Ans :

Some of the built-in exception are as follows,

(i) bad-alloc: This exception is thrown by new operator if it is unable to allocate memory for any variable.

(ii) overflow-error: This exception is thrown if overflow occurs while performing mathematical operations.

(iii) bad-cast: This exception is thrown by dynamic-cast if its result is not correct.

(iv) underflow-error: This exception is thrown if underflow occurs while performing mathematical operations.

(v) domain-error: This exception is thrown if any invalid domain is used while performing mathematical operations.

(vi) logic-error: This exception is theoretically identified while reading the code.

(vii) invalid-argument: This exception is thrown if the invalid arguments are encountered.

(viii) runtime-error: This exception is not theoretically identified while reading the code.

4.1.2 Throwing an Exception

Q3. Describe the role of try, throw and catch in exceptions.

Ans :

i) Try

Try is a keyword that is used to detect the exceptions i.e., run time errors. The statements that may cause exceptions are kept inside the try block.

Syntax

```
try
{
    //code
    throw exception;
}
```

A try block can throw more than one exception. There should be a catch block to handle the exceptions thrown by try block.

ii) Catch

The catch block handles an exception thrown by the try block. It defines actions to be taken when a run time error occurs.

Syntax

```
catch(type argument)
{
    //code
}
```

A catch block takes an argument as an exception. These arguments specify the type of an exception that can be handled by the catch block. When an exception is thrown the control goes to catch block. If the type of exception thrown matches the argument then the catch block is executed, otherwise the program terminates abnormally. If no exception is thrown from the try block then the catch block is skipped and control goes imme-diately to the next statement following the catch block.

iii) Throw

An exception detected in try block is thrown us-ing "throw" keyword. An exception can be thrown using throw statement in following number of ways,

throw (exception);

throw exception;

throw;

Where, 'exception' is an object of any type in-cluding a constant. The third form of throw statement is used in rethrowing an exception. The objects that are intended for error handling can also be thrown.

The point at which an exception is thrown is called as a throw point. When exception is thrown then the control leaves the try block and it reaches to the catch block associated with the try block where the exception is handled.

The throw point can be in a nested function call or in a nested scope within a try block. In any one of these cases the control is transferred to the catch statement.

4.1.3 Handling an Exception

Q4. How to handle an exception in C++.

Ans :

Exception Handling Mechanism

C++ exception handling mechanism is basically built upon three keywords, namely, try, throw, and catch. The keyword try is used to preface a block of statements (surrounded by braces) which may generate exceptions. This block of statements is known as try block. When an exception is detected, it is thrown using a throw statement in the try block. A catch block defined by the keyword catch 'catches' the exception 'thrown' by the throw statement in the try block, and handles it appropriately. The relationship is shown in Figure.

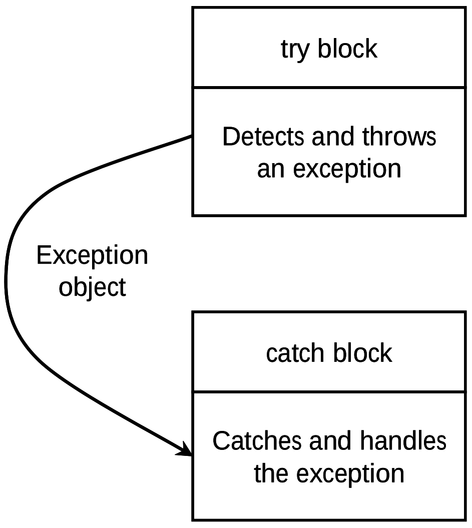

try block
Detects and throws an exception
Exception object
catch block
Catches and handles the exception

Fig.: The block throwing exception

The catch block that catches an exception must immediately follow the try block that throws the exception. The general form of these two blocks are as follows:

```
. . . . .
. . . . .
try
{
      . . . . .
      throw exception;            // Block of statements which
      . . . . .                   // detects and throws an exception
      . . . . .
}
catch(type arg)                   // Catches exception
{
      . . . . .                   // Block of statements that
      . . . . .                   // handles the exception
      . . . . .
      . . . . .
)
. . . . .
. . . . .
```

When the try block throws -an exception, the program control leaves the try block and enters the catch statement of the catch block. Note that exceptions are objects used to transmit information about a problem. If the type of object thrown matches the arg type in the catch statement, then catch block is executed for handling the exception. If they do not match, the program is aborted with the help of the abort() function which is invoked by default. When no exception is detected and thrown, the control goes to the statement immediately after the catch block. That is, the catch block is skipped. This simple try-catch mechanism is illustrated in Program.

Try Block Throwing An Exception

```
#include <iostream>

using namespace std;
int main()
{
      int a,b;
      cout << “Enter Values of a and b \n”;
      cin >> a;
      cin >> b;
      int x = a-b;
      try
      {
            if(x ! = 0)
            {
            cout << “Result(a/x) = “ << a/x << “\n”;
            }
```

```
          else                      // There is an exception
          {
                throw(x);           // Throws int object
          }
    }
    catch(int i)                    // Catches the exception
    {
                cout << "Exception caught: x = " << x << "\n";
    }
          cout << "END";
          return 0;
    }
```

The output of Program 13.1

First Run

```
Enter Values of a and b
20 15
Result(a/x) = 4
END
```

Second Run

```
Enter Values of a and b
10 10
Exception caught: x = 0
END
```

4.1.4 Object-Oriented Exception Handling with Classes

Q5. Write about Object-Oriented Exception Handling with Classes.

Ans :

C++ introduces an object-oriented approach to exception handling. This is done by throwing an exception class instead of throwing an exception. This is depicted in the below example.

Example

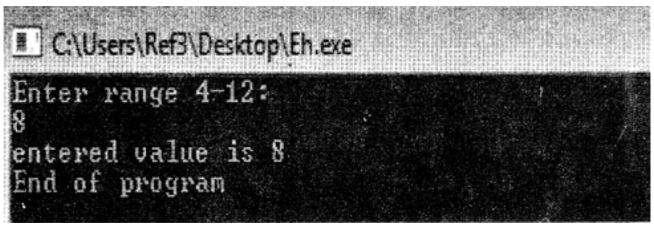
IntR.h

```
    #ifndef INTR_H
    #define INTR_H
    #include<iostream>
    using namespace std;
    class IntR
{
private:
    int input_value;
    int lower_limit;
    int upper_limit;
    public:
    class OutOfR
    {
        };
        IntR(int low, int high)
        {
                lower_limit = low;
                upper_limit = high;
        }
                int getInput()
                {
                        cin>>input_value;
                        if(input_value<lower_limit ||
                                input_value>upper_limit)
                        throw OutOfR();
                        return input_value;
                }
        };
        #endif
```

Example

```
    #include<iostream>
    #include "IntR.h"
    using namespace std;
    int main()
    {
        IntR range (4,12);
        int value;
        cout<<"Enter range 4-12 :\n";
        try
        {
                value = range.getInput();
                cout<<"entered value is"
                            <<value«endl;
        }
        catch(IntR::OutOfR)
        {
                cout<<"entered value is out of range\n";
        }
        cout<<"End of program\n";
        return 0;
    }
```

Output



```
C:\Users\Ref3\Desktop\Eh.exe
Enter range 4-12:
8
entered value is 8
End of program
```

In the above program, the getInput function retrieves the user input and compares it with upper limit and lowerlimit. If the value is less than lowerlimit or greater than upper_limit then it should throw an out of range exception. But, it throws exception class rather than throwing an exception in the form of character string or some other value. Here, the class thrown is empty. The throw statement will create an instance of this class and then throws it as an exception. The catch block will catch this exception and then knows the type of exception.

4.1.5 Multiple Exceptions

Q6. Define multiple exception. Write the syntax of multiple exceptions.

Ans :

C++ even permits a user to catch multiple exceptions. Inorder to do this, a single catch block is defined for catching all the exceptions thrown by using different throw statements. This catch block is of generic type.

Syntax

```
try
{
    // try section
}
catch (object 1)
{
    // catch section1
}
catch (object 2)
{
    // catch section2
}
. . . . . . . . . .
. . . . . . . . . .
{
    // catch section-n
}
```

Q7. Write a program to perform exception handling with multiple catch.

Ans :

```
#include<iostream.h>
#include<conio.h>
void test(int x)
{
try
{
if(x>0)
throw x;
else
throw 'x';
}
catch(int x)
{
cout<<"Catch a integer and that
                    integer is:"<<x;
}
catch(char x)
{
cout<<"Catch a character and that
                    character is:"<<x;
}
}
void main()
{
clrscr();
cout<<"Testing multiple catches\n:";
test(10);
test(0);
getch();
}
```

Output

```
Testing multiple catchesCatch a integer and that integer is: 10Catch a character and that character is: x
```

4.1.6 Extracting Data from the Exception Class

Q8. Explain how to use exception handling in classes with an example program

Ans :

Exceptions are an integral and unavoidable part of the operating system and programming. One way you can handle them is to create classes whose behaviors are prepared to deal with abnormal behavior. There are two main ways you can involve classes with exception handling routines: classes that are involved in exceptions of their own operations and classes that are specially written to handle exceptions for other classes.

Transferring Exceptions to Classes

You can create a class that is not specifically oriented towards exceptions, as any of the classes we have used so far. The simplest way to take care of exceptions in classes is to use any normal class and handle its exceptions. Such a class appears like one of the classes we have used already, except that exceptions of its abnormal behavior are taken care of. If concerned with exceptions, the minimum thing you can do in your program is to make it “aware’ of eventual exceptions. This can be taken care of by including transactions or other valuable processing in a try block, followed by a three-dot catch as in catch(...). The catch in this case is prepared to handle any exception that could occur.

Example

```
#include <iostream>
#include <iomanip>
using namespace std;
const double PriceShirt = 0.99;
const double PricePants = 1.75;
structTCleaningOrder
{
intNumberOfShirts;
intNumberOfPants;
intNumberOfMisc;
};
int main(intargc, char* argv[])
{
TCleaningOrder Order;
doubleTotalPriceShirts, TotalPricePants;
doublePriceMisc, TotalPriceMisc;
doubleTotalOrder;
cout<< " - Georgetown Cleaning Services -\n";
cout<< "  - Customer Order Processing -\n";
try {
cout<< "Number of\n";
cout<< "Shirts: ";
cin>>Order.NumberOfShirts;
cout<< "Pairs of Paints: ";
cin>>Order.NumberOfPants;
cout<< "Misc. Items(if none, type 0): ";
cin>>Order.NumberMisc;
// If there are miscalleanous items,...
if(Order.NumberOfMisc> 0)
{
// let the user determine the price of this misc item
cout<< "Enter the price of each miscellanous item: ";
cin>>PriceMisc;
TotalPriceMisc = Order.NumberOfMisc * PriceMisc;
}
else
TotalPriceMisc = 0.00;
TotalPriceShirts = Order.NumberOfShirts * PriceShirt;
TotalPricePants  =Order.NumberOfPants  * PricePants;
TotalOrder = TotalPriceShirts + TotalPricePants + TotalPriceMisc;
cout<<setiosflags(ios::fixed) <<setprecision(2);
cout<< " - Georgetown Cleaning Services -";
cout<< "\n -  Customer Receipt  -";
cout<<  "\n===================";
cout<<  "\n Item\tNumber\tPrice";
cout<< "\n————————————————";
cout<<  "\n Shirts\t" <<Order.NumberOfShirts
<< "\t$" <<TotalPriceShirts;
cout<<  "\n Pants\t" <<Order.NumberOfPants
<< "\t$" <<TotalPricePants;
cout<<  "\n Misc\t" <<Order.NumberOfMisc
<< "\t$" <<TotalPriceMisc;
cout<<  "\n===================";
cout<<  "\n Total Order:\t$" <<TotalOrder;
}
catch(...)
{
cout<<  "\nSomething went wrong - Too Bad";
}
return 0;
}
```

4.1.7 Re-throwing an Exception

Q9. What is re-throwing an exception?

Ans :

It is also possible to again pass the exception received by another exception handler. That is exception is thrown from the catch block. This is re-throwing an exception.

The following is the syntax for this:

Throw;

Throw with out any arguments is used for this.

When an exception is rethrown, it is propagated outward to the next catch block.

Q10. Write a program to demonstrate re-throwing the exception.

Ans :

```
#include <iostream>
using namespace std;
void MyHandler()
{
    try
    {
        throw "hello";
    }
    catch (const char*)
    {
    cout<<"Caught exception inside MyHandler\n";
    throw; //rethrow char* out of function
    }
}
int main()
{
    cout<< "Main start";
    try
    {
        MyHandler();
    }
    catch(const char*)
    {
      cout<<"Caught exception inside Main\n";
    }
        cout<< "Main end";
        return 0;
}
```

Output

```
Main start
Caught exception inside MyHandler
Caught exception inside Main
Main end
```

Thus, exception rethrown by the catch block inside MyHandler() is caught inside main();

4.2 Templates

4.2.1 Introduction

Q11. What is a template? Explain the need of a template?

Ans :

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as vector, but we can define many different kinds of vectors for example, vector <int> or vector <string>.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

Need of Template

Template allows user to process different data by declaring only a single function or class. The advantage, of using template in a program is that it overcomes the limitation that arises due to function overloading (i.e., increase in program length and creation of larger number of variable in memory) and adds flexibility to a program. A template can even extend the portability of classes. It allows user to process different data by declaring only a single function or class.

Q12. Define STL.

Ans :

The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provides general-purpose templatized classes and functions that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

At the core of the C++ Standard Template Library are following three well-structured components:

Component	Description
Containers	Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.
Algorithms	Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
Iterators	Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

4.2.2 Function Templates

Q13. What is function template? How to define it?

Ans :

Function Templates

A function template works in a similar to a normal function, with one key difference.

A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

Declare a function template

A function template starts with the keyword template followed by template parameter/s inside <> which is followed by function declaration.

```
template<class T>
T someFunction(T arg)
{
... .. ...
}
```

In the above code, T is a template argument that accepts different data types (int, float), and class is a keyword.

You can also use keyword typename instead of class in the above example.

When, an argument of a data type is passed to someFunction( ), compiler generates a new version of someFunction() for the given data type.

Q14. Write a C++ program to add two numbers using function template.

Ans :

```
#include <iostream>
#include <conio.h>
using namespace std;
template<class t1,class t2>
void sum(t1 a,t2 b) // defining template
                    // function
{
    cout<<"Sum="<<a+b<<endl;
}
int main()
{
   int a,b;
   float x,y;
   cout<<"Enter two integer data: ";
   cin>>a>>b;
   cout<<"Enter two float data: ";
   cin>>x>>y;
   sum(a,b); // adding two integer type data
   sum(x,y); // adding two float type data
   sum(a,x); // adding a float and integer type data
   getch();
   return 0;
}
```

This program illustrates the use of template function in C++. A template function *sum()* is created which accepts two arguments and add them. The type of argument is not defined until the function is called. This single function is used to add two data of integer type, float type and, integer and float type. We don't need to write separate functions for different data types. In this way, a single function can be used to process data of various type using function template.

Output

```
Enter two integer data: 6 10
Enter two float data: 5.8 3.3
Sum=16
Sum=9.1
Sum=11.8
```

Q15. Write a program to display largest among two numbers using function templates.

Ans :

// If two characters are passed to function template, character with larger ASCII value is displayed.

```
#include<iostream>
usingnamespace std;
      // template function
template<class T>
T Large(T n1, T n2)
{
return(n1 > n2)? n1 : n2;
}
```

```
int main()
{
int i1, i2;
float f1, f2;
char c1, c2;
cout <<"Enter two integers:\n";
cin >> i1 >> i2;
cout <<Large(i1, i2)<<" is larger."<< endl;
cout <<"\nEnter two floating-point numbers:\n";
cin >> f1 >> f2;
cout <<Large(f1, f2)<<" is larger."<< endl;
cout <<"\nEnter two characters:\n";
cin >> c1 >> c2;
cout <<Large(c1, c2)<<" has larger ASCII value.";
return0;
}
```

Output

```
Enter two integers:
5
10
10 is larger.
Enter two floating-point numbers:
12.4
10.2
12.4 is larger.
Enter two characters:
z
Z
z has larger ASCII value.
```

In the above program, a function template Large() is defined that accepts two arguments *n1* and *n2* of data type T. T signifies that argument can be of any data type.

Q16. Write a Program to swap data using function templates.

Ans :

```
#include <iostream>
using namespace std;
template <typename T>
void Swap(T &n1, T &n2)
{
```

```
T temp;
temp = n1;
n1 = n2;
n2 = temp;
}
int main()
{
int i1 = 1, i2 = 2;
float f1 = 1.1, f2 = 2.2;
char c1 = 'a', c2 = 'b';
cout << "Before passing data to function template.\n";
cout << "i1 = " << i1 << "\ni2 = " << i2;
cout << "\nf1 = " << f1 << "\nf2 = " << f2;
cout << "\nc1 = " << c1 << "\nc2 = " << c2;
Swap(i1, i2);
Swap(f1, f2);
Swap(c1, c2);
cout << "\n\nAfter passing data to function template.\n";
cout << "i1 = " << i1 << "\ni2 = " << i2;
cout << "\nf1 = " << f1 << "\nf2 = " << f2;
cout << "\nc1 = " << c1 << "\nc2 = " << c2;
return 0;
}
```

Output

```
Before passing data to function template.
i1 = 1
i2 = 2
f1 = 1.1
f2 = 2.2
c1 = a
c2 = b
After passing data to function template.
i1 = 2
i2 = 1
f1 = 2.2
f2 = 1.1
c1 = b
c2 = a
```

In this program, instead of calling a function by passing a value, a call by reference is issued. The Swap() function template takes two arguments and swaps them by reference.

Q17. Define template instantiation?

Ans :

The compiler creates functions using function templates.

```
int i = 2, j = 3;
cout << max(i, j);
string a(''Hello''), b(''World'');
cout<< max(a, b);
```

In this case, the compiler creates two different max()functions using the function template

template<typenameT> T max(constT& a, constT& b);

This is called template instantiation. The parameter T in the template definition is called the formal parameteror formal argument.

In the above code, the template is instantiated with the actual argumentsintand string, respectively.

4.2.2.1 Function Templates with Multiple Types

Q18. Explain about how Multiple Types para-meters are used with Function Templates.

Ans :

With templates, you may have more than one template-type parameters. It goes like:

template<class T1, class T2, ... >

Where T1 and T2 are type-names to the function template. You may use any other specific name, rather than T1, T2. Note that the usage of '**...**' above does **not** mean that this template

Let's have a simple example taking two template parameters:

```
Template<class T1, class T2>
void PrintNumbers(const T1& t1Data, const T2& t2Data)
{
cout <<"First value:"<< t1Data;
cout <<"Second value:"<< t2Data;
}
```

And we can simply call it as:

```
printNumbers(10, 100);          // int, int
PrintNumbers(14, 14.5); // int, double
PrintNumbers(59.66, 150);       // double, int
```

Another way of using the maxtemplate with arguments of different types is changing its definition in the following way:

```
template<typenameT1, typenameT2>
T1 max(constT1& a, constT2& b)
{
```

```
return (a > b) ? a : b;
}
void f() {
cout<< max(4, 5.5);// T1 isint, T2 isdouble
cout<< max(5.5, 4);// T1 isdouble, T2 isint
}
```

In similar fashion, function templates may have 3 or more type parameters, and each of them would map to the argument types specified in function call. As an example, the following function template is legal:

```
        template<class T1, class T2, class T3>
    T2 DoSomething(const T1 tArray[], T2 tDefaultValue, T3& tResult)
{
...
}
```

Q19. Write a program to implement function template with multiple arguments.

Ans :

The following is an example of a template supporting multiple types:

```
#include <iostream>
using namespace std;
    template <typename T, typename U>
    void squareAndPrint(T x, U y)
    {
    T result;
    U otherVar;
    cout << "X: " << x << " " << x * x
        << endl;
    cout << "Y: " << y << " " << y * y << endl;
    };
main()
{
    int   ii = 2;
    float jj = 2.1;
    squareAndPrint<int,float>(ii, jj);
    }
```

OUTPUT

```
X: 2 4
Y: 2.1 4.41
```

A single type can only be specified once.

Q20. What are Non-type template parameters demonstrate with an example.

Ans :

Non-type template parameters provide the ability to pass a constant expression at compile time. The constant expression may also be an address of a function, object or static class member.

The following is an example of a template function supporting a non-type parameter "count" used for the array size and loop count:

```
#include <iostream>
using namespace std;
template <typename T, int count>
void loopIt(T x)
{
T val[count];
for(int ii=0; ii<count; ii++)
{
val[ii] = x++;
cout <<  val[ii] << endl;
}
};
main()
{
float xx = 2.1;
    loopIt<float,3>(xx);
}
```

```
Compile: g++ test.cpp
Run: ./a.out
2.1
3.1
4.1
```

Q21. Specify a default type parameter and default non-type parameter with example program.

Ans :

```
#include <iostream>
using namespace std;
template <typename T=float, int count=3>
T multIt(T x)
{
for(int ii=0; ii<count; ii++)
{
        x = x * x;
        }
        return  x;
        };
        main()
        {
        float  xx = 2.1;
        cout << xx <<  ": "  << multIt<>(xx)
              << endl;;
        }
Compile:  g++ test.cpp -std=c++0x
Run:  ./a.out
2.1: 378.228
```

4.2.2.2 Overloading with Function Templates

Q22. Define Overloading with Template Function? Explain with an example program.

Ans :

If there are more than one function of same name in a program which differ only by number and/ or types of parameter, it is called function overloading. If at least one of these function is a template function, then it is called template function overloading. Template function can be overloaded either by using template functions or normal C++ functions of same name.

Example

C++ program to overload template function for sum of numbers.

```
#include <iostream>
#include <conio.h>
using namespace std;
template<class t1>
void sum(t1 a,t1 b,t1 c)
{
             cout<<"Template function 1: Sum = "<<a+b+c<<endl;
}
template <class t1,class t2>
void sum(t1 a,t1 b,t2 c)
{
        cout<<"Template function 2: Sum
                   = "<<a+b+c<<endl;
}
```

```
void sum(int a,int b)
{
        cout<<"Normal function: Sum
                      = "<<a+b<<endl;
}
int main()
{
   int a,b;
   float x,y,z;
   cout<<"Enter two integer data: ";
   cin>>a>>b;
   cout<<"Enter three float data: ";
   cin>>x>>y>>z;
   sum(x,y,z); // calls first template function
   sum(a,b,z); // calls first template function
   sum(a,b); // calls normal function
   getch();
   return 0;
}
```

In this program, template function is overloaded by using normal function and template function. Three functions named sum() are created. The first function accepts three arguments of same type. The second function accepts three argument, two of same type and one of different and, the third function accepts two arguments of int type. First and second function are template functions while third is normal function. Function call is made from main() function and various arguments are sent. The compiler matches the argument in call statement with arguments in function definition and calls a function when match is found.

Output

```
Enter two integer data: 5 9
Enter three float data: 2.3 5.6 9.5
Template function 1: Sum = 17.4
Template function 2: Sum = 23.5
Normal function: Sum = 14
```

4.2.3 Class Templates

4.2.3.1 Introduction

Q23. What is Class Template ? how to define it.

Ans :

Like function templates, you can also create class templates for generic class operations.

Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/function should be performed on all classes/ functions.

However, class templates make it easy to reuse the same code for all data types.

Q24. How to declare a class template?

Ans :

```
template<class T>
class className
{
... .. ...
public:
T var;
T someOperation(T arg);
... .. ...
};
```

In the above declaration, T is the template argument which is a placeholder for the data type used.

Inside the class body, a member variable *var* and a member function someOperation() are both of type T.

Q25. How to create a class template object?

Ans :

To create a class template object, you need to define the data type inside a <> when creation.

```
className<dataType> classObject;
```

For example

```
className<int> classObject;
className<float> classObject;
className<string> classObject;
```

Q26. Write a program to display Simple calculator using Class template.

Ans :

Program to add, subtract, multiply and divide two numbers using class template

```
#include<iostream>
usingnamespace std;
template<class T>
classCalculator
{
private:
T num1, num2;
public:
Calculator(T n1, T n2)
{
num1 = n1;
num2 = n2;
}

void displayResult()
{
cout <<"Numbers are: "<< num1 <<" and "<< num2 <<"."<< endl;
cout <<"Addition is: "<< add()<< endl;
cout <<"Subtraction is: "<< subtract()<< endl;
cout <<"Product is: "<< multiply()<< endl;
cout <<"Division is: "<< divide()<< endl;
}
T add(){return num1 + num2;}
T subtract(){return num1 - num2;}
T multiply(){return num1 * num2;}
T divide(){return num1 / num2;}
};
int main()
{
```

```
Calculator<int> intCalc(2,1);
Calculator<float> floatCalc(2.4,1.2);
cout <<"Int results:"<< endl;
intCalc.displayResult();
cout << endl <<"Float results:"<< endl;
floatCalc.displayResult();
return0;
}
```

Output

```
Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2
Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2
```

In the above program, a class template Calculator is declared.

The class contains two private members of type T: num1 & num2, and a constructor to initalize the members.

Q27. Write a C++ program to use class template.

Ans :

```
#include <iostream>
#include <conio.h>
using namespace std;
template<class t1,class t2>
class sample
{
   t1 a;
   t2 b;
   public:
      void getdata()
      {
         cout<<"Enter a and b: ";
         cin>>a>>b;
      }
      void display()
      {
         cout<<"Displaying values"<<endl;
         cout<<"a="<<a<<endl;
         cout<<"b="<<b<<endl;
      }
};
int main()
{
   sample<int,int> s1;
   sample<int,char> s2;
   sample<int,float> s3;
   cout <<"Two Integer data"<<endl;
   s1.getdata();
   s1.display();
   cout <<"Integer and Character data"<<endl;
   s2.getdata();
   s2.display();
   cout <<"Integer and Float data"<<endl;
   s3.getdata();
   s3.display();
   getch();
   return 0;
}
```

In this program, a template class *sample* is created. It has two data a and b of generic types and two methods: *getdata()* to give input and *display()* to display data. Three object s1, s2 and s3 of this class is created. s1 operates on both integer data, s2 operates on one integer and another character data and s3 operates on one integer and another float data. Since, *sample* is a template class, it supports various data types.

Output

```
Two Integer data
Enter a and b: 7 11
Displaying values
a=7
b=11
Integer and Character data
Enter a and b: 4 v
Displaying values
a=4
b=v
Integer and Float data
Enter a and b: 14 19.67
Displaying values
a=14
b=19.67
```

Q28. What are the Differences between class template and function template?

Ans :

C++ Function templates are those functions which can handle different data types without separate code for each of them. For a similar operation on several kinds of data types, a programmer need not write different versions by overloading a function. It is enough if we writes a C++ template based function

```
template <class T>
    T Add(T a, T b) //C++ function template
                    // sample
{
return a+b;
}
```

now t can be int, foat or any other data type.

so using function template we can work with many datatypes in one function.

we can call the function by using int x=Add(2,4)

4.2.3.2 Defining Objects of the Class Template

Q29. How class template can be instantiated.

(OR)

How to define objects for a class template

Ans :

A class template by itself is not a type, or an object, or any other entity. No code is generated from a source file that contains only template definitions. In order for any code to appear, a template must be instantiated: the template arguments must be provided so that the compiler can generate an actual class.

This can be done in two ways

i) Explicit Instantiation

ii) implicit Instantiation

i) Explicit instantiation

An explicit instantiation definition forces instantiation of the class, struct, or union they refer to.

An explicit instantiation declaration (an extern template) prevents implicit instantiations: the code that would otherwise cause an implicit instantiation has to use the explicit instantiation definition provided somewhere else in the program.

Classes, functions, variables, and member template specializations can be explicitly instantiated from their templates.

Member functions, member classes, and static data members of class templates can be explicitly instantiated from their member definitions.

Explicit instantiation can only appear in the enclosing namespace of the template, unless it uses qualified-id:

Example

```
namespace N
{
template<class T>class Y
{void mf()
{
}
};// template definition
}
        // template class Y<int>; // error: class template Y not visible in the global namespace using N::Y;
        // template class Y<int>; // error: explicit instantiation outside
        // of the namespace of the template
        templateclass N::Y<char*>;// OK: explicit instantiation
        templatevoid N::Y<double>::mf();// OK: explicit instantiation
```

Explicit instantiation definitions ignore member access specifiers: parameter types and return types may be private.

ii) Implicit instantiation

When code refers to a template in context that requires a completely defined type, or when the completeness of the type affects the code, and this particular type has not been explicitly instantiated, implicit instantiation occurs.

For example, when an object of this type is constructed, but not when a pointer to this type is constructed.

This applies to the members of the class template: unless the member is used in the program, it is not instantiated, and does not require a definition.

```
template<class T>struct Z {
void f(){}
void g();// never defined
};// template definition
templatestruct Z<double>;// explicit instantiation of Z<double>
Z<int> a;// implicit instantiation of Z<int>
Z<char>* p;// nothing is instantiated here
p->f();// implicit instantiation of Z<char> and Z<char>::f() occurs here.
// Z<char>::g() is never needed and never instantiated: it does not have to be defined
```

4.2.3.3 Class Templates and Inheritance

Q30. Explain about how Class Templates can use the concept of Inheritance.

Ans :

Templates and inheritance are used to write a code so that various forms of it can be created. They help in deriving new types from the existing ones.

The following are the two relationships of template and inheritance.

(i) Template Class Derived from a Non-template Class

A template can be derived from non-template classes to provide a common implementation for a group of templates. Consider the following example,

```
template<class T>
class example<T*>: private example <void*>
{
//statements
};
```

(ii) Template Class Derived from another Template Class

The members of a base class can be used in the implementation of the derived classes. But, when the template parameter of a derived class is used in the implementation of a base class then the base class should also be parameterized.

Consider the following example,

```
template<class T>
class example
{
//statements
};
template<class T>
class sca:public example<T>
{
//statements
};
```

Though the base and derived classes consist of same template parameter, this technique is used very rarely. Instead of this, a technique in which the derived types are passed to the base class is mostly used.

Example

```
template <class C>
class ArithmaticOperations
{
public:
    bool operator + (const C&) const;
    bool operator – (const C&) const;
    const C& derived() const
    {
        return staticCasting <const C& > (*this);
    }
};
template<class T>
classOpsContainer:publicBasicOperations
<OpsContainer<T>>
{
    public:
        sizeOfcontainer size() const;
        T&operator[ ] (sizeofcontainer);
        const T&operator [ ] (sizeOfcontainer) const;
};
```

Short Question & Answers

1. What is an Exceptions?

Ans :

Exceptions are the run-time errors that occurs during the program execution.

Exception occurs due to,

(i) Division-by-zero condition

(ii) Exceeding the bounds of an array

(iii) Running out of memory

(iv) Falling short of memory

(v) Object initialization to an impossible value.

These exceptions can be handled in a systematic way by using three keywords try, throw and catch.

2. List out various built-in exceptions.

Ans :

Some of the built-in exception are as follows,

(i) bad-alloc: This exception is thrown by new operator if it is unable to allocate memory for any variable.

(ii) overflow-error: This exception is thrown if overflow occurs while performing mathematical operations.

(iii) bad-cast: This exception is thrown by dynamic-cast if its result is not correct.

(iv) underflow-error: This exception is thrown if underflow occurs while performing mathematical operations.

(v) domain-error: This exception is thrown if any invalid domain is used while performing mathematical operations.

(vi) logic-error: This exception is theoretically identified while reading the code.

(vii) invalid-argument: This exception is thrown if the invalid arguments are encountered.

(viii) runtime-error: This exception is not theoretically identified while reading the code.

3. What is re-throwing an exception?

Ans :

It is also possible to again pass the exception received by another exception handler. That is exception is thrown from the catch block. This is re-throwing an exception.

The following is the syntax for this:

Throw;

Throw with out any arguments is used for this.

When an exception is rethrown, it is propagated outward to the next catch block.

4. What is a template? Explain the need of a template?

Ans :

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as vector, but we can define many different kinds of vectors for example, vector <int> or vector <string>.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

Need of Template

Template allows user to process different data by declaring only a single function or class. The advantage, of using template in a program is that it overcomes the limitation that arises due to function overloading (i.e., increase in program length and creation of larger number of variable in memory) and adds flexibility to a program. A template can even extend the portability of classes. It allows user to process different data by declaring only a single function or class.

5. What is function template? How to define it?

Ans :

A function template works in a similar to a normal function, with one key difference.

A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

6. Define template instantiation?

Ans :

The compiler creates functions using function templates.

```
int i = 2, j = 3;
cout << max(i, j);
string a(''Hello''), b(''World'');
cout<< max(a, b);
```

In this case, the compiler creates two different max()functions using the function template

template<typenameT> T max(constT& a, constT& b);

This is called template instantiation. The parameter T in the template definition is called the formal parameteror formal argument.

In the above code, the template is instantiated with the actual argumentsintand string, respectively.

7. Overloading with Template Function?

Ans :

If there are more than one function of same name in a program which differ only by number and/ or types of parameter, it is called function overloading. If at least one of these function is a template function, then it is called template function overloading. Template function can be overloaded either by using template functions or normal C++ functions of same name.

8. Explicit instantiation

Ans :

An explicit instantiation definition forces instantiation of the class, struct, or union they refer to.

An explicit instantiation declaration (an extern template) prevents implicit instantiations: the code that would otherwise cause an implicit instantiation has to use the explicit instantiation definition provided somewhere else in the program.

Classes, functions, variables, and member template specializations can be explicitly instantiated from their templates.

Member functions, member classes, and static data members of class templates can be explicitly instantiated from their member definitions.

9. What is Class Template ?

Ans :

Like function templates, you can also create class templates for generic class operations.

Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/ function should be performed on all classes/functions.

However, class templates make it easy to reuse the same code for all data types.

10. How to declare a class template?

Ans :

```
template<class T>
class className
{
... .. ...
public:
T var;
T someOperation(T arg);
... .. ...
};
```

In the above declaration, T is the template argument which is a placeholder for the data type used.

Inside the class body, a member variable *var* and a member function someOperation() are both of type T.

Multiple Choice Questions

1. Which header file is used to declare the standard exception? [c]
   (a) #include<exception> (b) #include<except>
   (c) #include<error> (d) none of the mentioned
2. What are the perdefined exceptions in c++? [a]
   (a) Memory allocation errors (b) I/O errors
   (c) both a & b (d) None of the mentioned
3. Which of the following problem causes an exception? [d]
   (a) Missing semicolon in statement in main().
   (b) A problem in calling function.
   (c) A syntax error.
   (d) A run-time error.
4. How to declare a template? [c]
   a) tem b) temp
   c) template<> d) none of the mentioned
5. How many types of templates are there in c++? [b]
   (a) 1 (b) 2
   (c) 3 (d) 4View
6. What is meant by template parameter? [a]
   (a) It can be used to pass a type as argument
   (b) It can be used to evaluate a type.
   (c) It can of no return type
   (d) None of the mentioned
7. Among the following what is the type of the template [c]
   (a) class (b) function
   (c) both a & b (d) typename
8. Which is used to describe the function using placeholder types? [b]
   (a) template parameters (b) template type parameters
   (c) template type (d) none of the mentioned
9. Templates are processed by __________ [c]
   (a) Loader (b) Linker
   (c) Compiler (d) Assembler
10. Templates are used for which data types? [a]
    (a) any data type (b) basic data type
    (c) derived Data type (d) user defined Data Type

Fill in the blanks

1. ________ keyword is used to handle the expection?
2. ________ is used to throw a exception?
3. ________ and ________ are the keywords can be used in template?
4. ________ is used to check the error in the block?
5. ________ should present when throwing a object?
6. ________ is dependant on template parameter?
7. In ________ place, is the validity of template parameters?
8. For ________ and ________ we use :: template-template parameter?
9. Templates support ________ compilation
10. A class created from a template is called ________

ANSWERS

1. catch
2. throw
3. class and function
4. try
5. copy constructor
6. base class
7. inside that block only
8. binding and rebinding
9. on-demand
10. template class

LAB PROGRAMMES

Q1. Write a program to print the sum of digits of a given number.

Ans :

```
#include<iostream>
using namespace std;
int main()
{
        unsigned long i,p,n,sum=0;
        cout<<"Enter any number:";
        cin>>n;

        while(n!=0)
        {
                p=n%10;
                sum+=p;
                n=n/10;
        }
                cout<<endl<<"Sum of digits is:"<<sum;
        return 0;
}
```

Output

```
Enter any number:361
Sum of digits is:10
```

Q2. Write a program to check whether the given number is Armstrong or not.

Ans :

```
#include <iostream>
using namespace std;
int main()
{
  int origNum, num, rem, sum = 0;
  cout << "Enter a positive  integer: ";
  cin >> origNum;
  num = origNum;
  while(num != 0)
   {
      digit = num % 10;
      sum += digit * digit * digit;
      num /= 10;
   }
if(sum == origNum)
        cout << origNum << " is an Armstrong number.";
  else
        cout << origNum << " is not an Armstrong number.";
  return 0;
}
```

Output

```
Enter a positive integer: 371
371 is an Armstrong number.
```

In the above program, a positive integer is asked to enter by the user which is stored in th

Q3. Write a program to check whether the given string is Palindrome or not.

Ans :

```
#include<iostream>
usingnamespace std;
int main()
{
int n, num, digit, rev =0;
    cout <<"Enter a positive number: ";
     cin >> num;
     n = num;
do
{
         digit = num %10;
        rev =(rev *10)+ digit;
         num = num /10;
}while(num !=0);
          cout <<" The reverse of the number is: "<< rev
           << endl;
if(n == rev)
        cout <<" The number is a palindrome";
else
        cout <<" The number is not a palindrome";
return0;
}
```

Output

```
Enter a positive number: 12321
The reverse of the number is: 12321
The number is a palindrome
Enter a positive number: 12331
The reverse of the number is: 13321
The number is not a palindrome
```

Q4. Write a program to read the student name, roll no, marks and display the same using class and object.

Ans :

```
#include<iostream.h>
#include<stdio.h>
#include<dos.h>
class student
{
        int roll;
        char name[25];
        char add [25];
        char *city;
        public: student()
        {
                cout<<"welcome in the student
                            information system"<<endl;
        }
        void getdata()
        {
                cout<<"\n enter the student roll no.";
                cin>>roll;
                cout<<"\n enter the student name";
                cin>>name;
                cout<<\n enter ther student address";
                cin>>add;
                cout<<"\n enter the student city";
                cin>>city;
        }
        void putdata()
        {
                cout<,"\n the student roll no:"<<roll;
                cout<<"\n the student name:"<<name;
                cout<<"\n the student coty:"<<city;
        }
};
class mrks: public student
{
        int sub1;
        int sub2;
        int sub3;
        int per;
        public: void input()
        {
                getdata();
                cout<<"\n enter the marks1:"
                cin>>sub1:
                cout<<"\n enter the marks2:";
                cin>>sub2;
                cout<<\n enter the marks3:";
                cin>>sub3;
        }
        void output()
        {
                putdata();
                cout<<"\n marks1:"<<sub1;
                cout<<"\n marks2:"<<sub2;
                cout<<"\n marks3:"<<sub3;
        }
        void calculate ()
        {
                per= (sub1+sub2+sub3)/3;
                cout<<"\n tottal percentage"<<per;
        }
};

void main()
{
        marks m1[25];
        int ch;
        int count=0;
        do
        {
                cout<<\n1.input data";
                cout<<\n2.output data";
                cout<<\n3. Calculate percentage";
                cout<<\n4.exit";
                cout<<\n enter the choice";
                cin>>ch;
                switch (ch)
                {
                    case 1:
                    m1.input();
                    count++;
                    break;

                  case2:
                    m1.output();
                    break;

                    case3:
                    m1.calculate();
                    break;
             }
        } while (ch!=4);
}
```

Q5. Write a program to find area of a rectangle, circle, and square using class and object.

Ans :

```
usingnamespacestd;
intarea(int);
intarea(int,int);
floatarea(float);
floatarea(float,float);
intmain()
{
        ints,l,b;
        floatr,bs,ht;
        cout<<"Enter side of a square:";
        cin>>s;
        cout<<"Enter length and breadth of rectangle:";
        cin>>l>>b;
        cout<<"Enter radius of circle:";
        cin>>r;
        cout<<"Enter base and height of triangle:";
        cin>>bs>>ht;
        cout<<"Area of square is"<<area(s);
        cout<<"\nArea of rectangle is "<<area(l,b);
        cout<<"\nArea of circle is "<<area(r);
        cout<<"\nArea of triangle is "<<area(bs,ht);
}
intarea(ints)
{
       return(s*s);
}
intarea(intl,intb)
{
       return(l*b);
}
floatarea(floatr)
{
       return(3.14*r*r);
}
floatarea(floatbs,floatht)
{
    return((bs*ht)/2);
}
```

Sample Input

```
Enter side of a square:2
Enter length and breadth of rectangle:3 6
Enter radius of circle:3
Enter base and height of triangle:4 4
```

Sample Output

```
Area of square is4
Area of rectangle is 18
Area of circle is 28.26
Area of triangle is 8
```

Q6. Write a program to implement inline function inside and outside of a class for.

Ans :

a) Finding the area of a square

b) Finding the area of a cube

```
#include<iostream.h>
#include<conio.h>
class power
{
public:
inline int square(int n)
{
return n*n;
}
inline int cube(int n)
{
return n*n*n;
}
};
void main()
{
int n,r;
power p;
clrscr();
cout<<"\nEnter the Number: \n" ;
cin>>n;
r=p.square(n);
cout<<"\nSquare of "<<n<<" = "<<r<<endl;
r=p.cube(n);
cout<<"\nCube of "<<n<<" = "<<r<<endl;
getch();
}
```

Q7. Write a program to implement friend function and friend class.

Ans :

The below example will completely illustrate the use of friend functions in C++ programming.

For Example

```
#include<iostream.h>
#include<conio.h>
classB;
classA
{
private:
inta;
public:
A()
{
a=25;
}
friendvoidshow(A,B);
};
classB
{
private:
intb;
public:
B()
{
b=35;
}
friendvoidshow(A,B);
};
voidshow(A x, B y)
{
intr;
r= x.a + y.b;
cout<<"The value of classA object ="<<x.a<<endl;
cout<<"The value of classB object ="<<y.b<<endl;
cout<<"The sum of both values ="<<r<<endl;
}
main()
{
A obj1;
B obj2;
show(obj1, obj2);
getch();
}
```

Code for Program to illustrate the use of friend classes in C++ Programming

```
#include<iostream.h>
#include<conio.h>
/******************************************///———
CLass Declarations ——————————
//*************************************/class beta;
/******************************************///———
—————— alpha ——————————
///*************************************/class alpha
{
private:
int alpha_data;

public:
      alpha()
         {
           alpha_data=0;
         }

      alpha(int d)
         {
           alpha_data=d;
         }

void show()
         {
           cout<<"\n Value of alpha_data =
                         "<<alpha_data<<endl;
         }
      friend beta;
   };
```

```
/**************************************///————————
————————— beta ——————————-——————————
///**************************************/class beta
{
private:
int beta_data;
public:
        beta()
            {
              beta_data=0;
            }
void show(alpha a)
            {
              cout<<" Value of beta_data =
                      "<<a.alpha_data<<endl;
            }
    };
 main()
    {
      clrscr();
      alpha a(786);
      beta b;
      a.show();
      b.show(a);
      getch();
return 0;
    }
```

Q8. Write a program to implement constructor and destructor with in a class.

Ans :

```
#include<iostream.h>
#include<conio.h>
class stu
{
                private: char name[20],add[20];
                         int roll,zip;
                public: stu ( );//Constructor
                         ~stu( );//Destructor
                         void read( );
                         void disp( );
};
stu :: stu( )
{
          cout<<"This is Student Details"<<endl;
}
void stu :: read( )
{
          cout<<"Enter the student Name";
          cin>>name;
          cout<<"Enter the student roll no ";
          cin>>roll;
          cout<<"Enter the student address";
          cin>>add;
          cout<<"Enter the Zipcode";
          cin>>zip;
}
void stu :: disp( )
{
          cout<<"Student Name :"<<name<<endl;
          cout<<"Roll no   is       :"<<roll<<endl;
          cout<<"Address is       :"<<add<<endl;
          cout<<"Zipcode is       :"<<zip;
}
stu : : ~stu( )
{
          cout<<"Student Detail is Closed";
}

void main( )
{
          stu s;
          clrscr( );
s.read ( );
s.disp ( );
getch( );
}
```

Output

```
Enter the student Name
James
Enter the student roll no
01
Enter the student address
Newyork
Enter the Zipcode
919108
Student Name : James
Roll no is : 01
```

Q9. Write a program to demonstrate hierarchical inheritance.

Ans :

C++ program to create Employee and Student inheriting from Person using Hierarchical Inheritance

```
#include <iostream>
#include <conio.h>
using namespace std;
class person
{
   char name[100],gender[10];
    int age;
    public:
        void getdata()
         {
            cout<<"Name: ";
           fflush(stdin);  /*clears input stream*/
            gets(name);
            cout<<"Age: ";
            cin>>age;
            cout<<"Gender: ";
            cin>>gender;
         }
        void display()
         {
           cout<<"Name: "<<name<<endl;
           cout<<"Age: "<<age<<endl;
          cout<<"Gender: "<<gender<<endl;
         }
};
class student: public person
{
   char institute[100], level[20];
    public:
        void getdata()
         {
           person::getdata();
           cout<<"Name of College/School: ";
            fflush(stdin);
            gets(institute);
            cout<<"Level: ";
            cin>>level;
         }
        void display()
         {
            person::display();
                cout<<"Name of College/School: "<<institute<<endl;
            cout<<"Level: "<<level<<endl;
         }
};
class employee: public person
{
   char company[100];
    float salary;
    public:
        void getdata()
         {
            person::getdata();
            cout<<"Name of Company: ";
             fflush(stdin);
            gets(company);
             cout<<"Salary: Rs.";
             cin>>salary;
         }
        void display()
         {
            person::display();
                 cout<<"Name of Company: "<<company<<endl;
            cout<<"Salary: Rs."<<salary<<endl;
         }
};
int main()
{
    student s;
   employee e;
   cout<<"Student"<<endl;
   cout<<"Enter data"<<endl;
    s.getdata();
   cout<<endl<<"Displaying data"<<endl;
    s.display();
   cout<<endl<<"Employee"<<endl;
   cout<<"Enter data"<<endl;
    e.getdata();
   cout<<endl<<"Displaying data"<<endl;
    e.display();
    getch();
    return 0;
}
```

Output

```
Student
Enter data
Name: John Wright
Age: 21
Gender: Male
Name of College/School: Abc Academy
Level: Bachelor
Displaying data
Name: John Wright
Age: 21
Gender: Male
Name of College/School: Abc Academy
Level: Bachelor
Employee
Enter data
Name: Mary White
Age: 24
Gender: Female
Name of Company: Xyz Consultant
Salary: $29000
Displaying data
Name: Mary White
Age: 24
Gender: Female
Name of Company: Xyz Consultant
Salary: $29000
```

10. Write a program to demonstrate multiple inheritances.

Ans :

```
#include<iostream.h>
#include<conio.h>
class student
{
    protected:
      int rno,m1,m2;
    public:
              void get()
             {
                cout<<"Enter the Roll no :";
                cin>>rno;
                cout<<"Enter the two marks    :";
                       cin>>m1>>m2;
             }
};
class sports
{
    protected:
      int sm;                  // sm = Sports mark
    public:
             void getsm()
            {
              cout<<"\nEnter the sports mark :";
              cin>>sm;

            }
};
class statement:public student,public sports
{
    int tot,avg;
    public:
    void display()
            {
              tot=(m1+m2+sm);
              avg=tot/3;
              cout<<"\n\n\tRoll No   :
                "<<rno<<"\n\tTotal       : "<<tot;
            cout<<"\n\tAverage     : "<<avg;
            }
};
void main()
{
   clrscr();
   statement obj;
   obj.get();
   obj.getsm();
   obj.display();
   getch();
}
```

Output:

```
      Enter the Roll no: 100
      Enter two marks
            90
            80
      Enter the Sports Mark: 90
            Roll No: 100
            Total      : 260
            Average: 86.66
```

Q11. Write a program to demonstrate the constructor overloading.

Ans :

```
/*  Example Program For Simple Example Program Of
Constructor Overloading In C++
    little drops @ thiyagaraaj.com
    Coded By:THIYAGARAAJ MP  */
#include<iostream>
#include<conio.h>
using namespace std;
class Example        {
    // Variable Declaration
    int a,b;
    public:
    //Constructor wuithout Argument
    Example()          {
    // Assign Values In Constructor
    a=50;
    b=100;
    cout<<"\nIm Constructor";
    }
    //Constructor with Argument
    Example(int x,int y)          {
    // Assign Values In Constructor
    a=x;
    b=y;
    cout<<"\nIm Constructor";
    }
    void Display()     {
    cout<<"\nValues :"<<a<<"\t"<<b;
    }
};
int main()                 {
        Example Object(10,20);
        Example Object2;
        // Constructor invoked.
        Object.Display();
        Object2.Display();
        // Wait For Output Screen
        getch();
        return 0;
}
```

Sample Output

```
Im Constructor
Im Constructor
Values :10      20
Values :50      100
```

Q12. Write a program to demonstrate static polymorphism.

Ans :

```
public Class StaticDemo
{
public void display(int x)
{
Console.WriteLine("Area of a Square:"+x*x);
}
public void display(int x, int y)
{
Console.WriteLine("Area of a Square:"+x*y);
}
public static void main(String args[])
{
StaticDemo spd=new StaticDemo();
Spd.display(5);
Spd.display(10,3);
}
}
```

Q13. Write a program to demonstrate dynamic polymorphism.

Ans :

```
Class BaseClass
{
Public void show ()
{
Console.WriteLine("From base class show method");
}
}
Public Class DynamicDemo : BaseClass
{
Public void show()
{
Console.WriteLine("From Derived Class show method");
}
Public static void main(String args[])
{
DynamicDemo dpd=new DynamicDemo ();
Dpd.show();
}
}
```

Q14. Write a program to implement polymorphism using pure virtual functions.

Ans :

```
#include<iostream.h>
#include<conio.h>
class base
{
     public:
        virtual void show()
        {
                   cout<<"\n   Base class show:";
        }
        void display()
        {
                 cout<<"\n   Base class display:" ;
        }
};
  class drive:public base
{
    public:
        void display()
        {
                cout<<"\n   Drive class display:";
        }
        void show()
        {
                cout<<"\n   Drive class show:";
        }
};
void main()
{
    clrscr();
    base obj1;
    base *p;
    cout<<"\n\t P points to base:\n"   ;
      p=&obj1;
    p->display();
    p->show();
      cout<<"\n\n\t P points to drive:\n";
    drive obj2;
    p=&obj2;
    p->display();
    p->show();
    getch();
}
```

Output:

```
                    P points to Base
                      Base class display
                    Base class show
                      P points to Drive
                      Base class Display
                    Drive class Show
```

Q15. Write a program to demonstrate the function templates and class templates.

Ans :

Function templates

```
#include<iostream.h>
#include<conio.h>
template<class t>
void swap(t &x,t &y)
{
    t temp=x;
    x=y;
    y=temp;
}
void fun(int a,int b,float c,float d)
{
    cout<<"\na and b before swaping
                :"<<a<<"\t"<<b;
    swap(a,b);
    cout<<"\na and b after swaping
                  :"<<a<<"\t"<<b;
    cout<<"\n\nc and d before swaping
                :"<<c<<"\t"<<d;
    swap(c,d);
    cout<<"\nc and d after swaping
                  :"<<c<<"\t"<<d;
}
void main()
{
        int a,b;
        float c,d;
        clrscr();
        cout<<"Enter A,B values(integer):";
        cin>>a>>b;
        cout<<"Enter C,D values(float):";
        cin>>c>>d;
        fun(a,b,c,d);
        getch();
}
```

Output:

```
Enter A, B values (integer): 10   20
Enter C, D values (float):      2.50   10.80
A and B before swapping: 10 20
A and B after swapping:   20 10
C and D before swapping: 2.50   10.80
C and D after swapping: 10.80   2.50
```

Class templates

//C++_Class_Templates.cpp

```
#include <iostream.h>
#include <vector>
template <typename T>
class MyQueue
{
std::vector<T> data;
public:
void Add(T const &);
void Remove();
void Print();
};
template <typename T> void MyQueue<T> ::Add(T const &d)
{
data.push_back(d);
}
template <typename T> void MyQueue<T>::Remove()
{
data.erase(data.begin( ) + 0,data.begin( ) + 1);
}
template <typename T> void MyQueue<T>::Print()
{
std::vector <int>::iterator It1;
It1 = data.begin();
for ( It1 = data.begin( ) ; It1 != data.end( ) ; It1++ )
cout << " " << *It1<<endl;
}
//Usage for C++ class templates
void main()
{
MyQueue<int> q;
q.Add(1);
q.Add(2);
cout<<"Before removing data"<<endl;
q.Print();
.Remove();
cout<<"After removing data"<<endl;
q.Print();
}
```

Q16. Write a program to demonstrate exception handling using try, catch, and finally.

Ans :

```
#include <iostream>
#include <string>
using namespace std;
int main() {
        int numerator, denominator, result;
        cout <<"Enter the Numerator:";
        cin>>numerator;
        cout<<"Enter the denominator:";
        cin>>denominator;
        try {
                if(denominator == 0) {
                        throw denominator;
                } else if (denominator < 0) {
                        throw "Negative denominator not
                                allowed";
                }
                result = numerator/denominator;
                cout<<"\nThe result of division is:"
                        <<result;
        }
        catch(int num) {
                cout<<"You cannot enter "<<num<<"
                                in denominator.";
        }
        catch (char* message) {
                cout<<message;
        }
}
```

FACULTY OF SCIENCE

B.Sc II - Semester (CBCS) Examination,

May/June-2019

PROGRAMMING IN C++

Time : 3 Hours] [Max. Marks : 80

PART - A (5 × 4 = 20 Marks)

(Short Answer Type)

Note: Answer any FIVE of the following questions.

1. Explain inline function with a program. **(Unit-I, SQA - 7)**
2. Write down the applications of OOP. **(Unit-II, SQA - 2)**
3. What is aggregation? List out the types of aggregation. **(Out of Syllabus)**
4. What is the need for private members? Explain. **(Unit-II, SQA - 6)**
5. State the rules associated with virtual functions. **(Unit-III, SQA - 12)**
6. Explain the different types of stream classes in C++. **(Unit-III, SQA - 30)**
7. What is a template? Write the syntax of a template. **(Unit-IV, SQA - 4)**
8. What is an exception? List out built-in exceptions. **(Unit-IV, SQA - 1, 2)**

PART - B (4 × 15 = 60 Marks)

(Essay Answer Type)

Note: Answer ALL the questions

9. (a) Explain the concept of binary search with proper example and program. **(Unit-I, Q.No. 39)**

OR

(b) What is function prototype? Discuss the process of passing data by value with proper program. **(Unit-I, Q.No. 53, 55)**

10. (a) What is a constructor? Explain about copy constructor with example program. **(Unit-II, Q.No. 19, 23)**

OR

(b) Explain instance and static members of a class with proper examples. **(Unit-II, Q.No. 31, 32)**

11. (a) What is polymorphism? Explain different types of polymorphism with examples. **(Unit-III, Q.No. 22)**

OR

(b) What are streams? Explain different formatted and unformatted I/O operations. **(Unit-III, Q.No. 30, 31, 32)**

12. (a) Explain about catching multiple exceptions with suitable program. **(Unit-IV, Q.No. 6, 7)**

OR

(b) What is function template? Write a C++ program to swap two numbers using function template. **(Unit-IV, Q.No. 13, 16)**

FACULTY OF SCIENCE

B.Sc II - Semester (CBCS) Examination,

May/June-2018

PROGRAMMING IN C++

Time : 3 Hours] [Max. Marks : 80

PART - A (5 × 4 = 20 Marks)

(Short Answer Type)

Note: Answer any FIVE of the following questions.

1. Explain about any two looping statements in C++ with proper example. **(Unit-I, SQA - 5)**
2. Define function overloading. Write a program to illustrate the same. **(Unit-I, SQA - 6)**
3. What are constructors? Write about types of constructors. **(Unit-II, SQA - 7, 8)**
4. What is a class? How classes provide data encapsulation? **(Unit-II, SQA - 4)**
5. What are the various access specifiers in C++? **(Unit-III, SQA - 4, 5, 6)**
6. Explain about different C++ unformatted I/O operations. **(Unit-III, SQA - 15)**
7. Describe about rethrowing an exception in C++. **(Unit-IV, SQA - 3)**
8. Explain the overloading with function template. **(Unit-IV, SQA - 7)**

PART - B (4 × 15 = 60 Marks)

(Essay Answer Type)

Note: Answer ALL the questions

9. (a) Explain concepts of object oriented programming with proper examples. **(Unit-II, Q.No. 3)**

OR

(b) What is an array? Differentiate one and two dimensional arrays. Write a C++ program to sort the given array of numbers in ascending order. **(Unit-I, Q.No. 32, 33)**

10. (a) Explain operator overloading as a concept of polymorphism. Write a program to overload operator + for adding two complex numbers. **(Unit-II, Q.No. 44)**

OR

(b) Write in detail about friend functions and friend classes with proper programs. **(Unit-II, Q.No. 34, 38, 39)**

11. (a) Write about inheritance and different types of inheritance in C++? Write a program to demonstrate the multiple inheritance. **(Unit-III, Q.No. 1, 3, 6)**

OR

(b) What are stream classes? Explain formatted I/O operations with example program. **(Unit-III, Q.No. 30, 32)**

12. (a) What is an exception? What happens if exceptions are not handled? Explain exception handling in C++. **(Unit-IV, Q.No. 1, 4)**

OR

(b) Write about templates in C++. Write a C++ program to demonstrate class template. **(Unit-IV, Q.No. 11, 26)**

FACULTY OF SCIENCE

B.Sc II - Semester (CBCS) Examination,

May/June-2017

PROGRAMMING IN C++

Time : 3 Hours] [Max. Marks : 80

PART - A (5 × 4 = 20 Marks)

(Short Answer Type)

Note: Answer any FIVE of the following questions.

1. What is a function overloading? Explain. **(Unit-I, SQA - 6)**
2. List and explain the operators those are used only in C++. **(Unit-I, SQA - 3)**
3. Explain about private member function. **(Unit-II, SQA - 10)**
4. Explain the types of aggregation with examples. **(______)**
5. Explain unformatted I/O functions. **(Unit-III, SQA - 15)**
6. What is Inheritance? Explain the types of inheritance. **(Unit-III, SQA - 2)**
7. What is an exception? Explain some built-in exceptions. **(Unit-IV, SQA - 1, 2)**
8. Define template. What is the need of a template. **(Unit-IV, SQA - 4)**

PART - B (4 × 15 = 60 Marks)

(Essay Answer Type)

Note: Answer ALL the questions

9. (a) What is searching? Explain the concepts of binary search with an example program. **(Unit-I, Q.No. 39)**

OR

(b) What is an array? Write a C++ program to multiplication of two matrices. **(Unit-I, Q.No. 32, 37)**

10. (a) What is over loading? Explain operator overloading with an example program. **(Unit-II, Q.No. 44, 45, 47)**

OR

(b) Discuss about object conversion with an example program. **(______)**

11. (a) What is a polymorphism? Explain different type of polymorphism with example program. **(Unit-III, Q.No. 22)**

OR

(b) Explain the concept of virtual function with program. **(Unit-III, Q.No. 23, 25)**

12. (a) Write about object oriented exception handling with classes. **(Unit-IV, Q.No. 5)**

OR

(b) Explain function template with an example program. **(Unit-IV, Q.No. 13, 14)**

FACULTY OF SCIENCE

B.Sc. I-Year II-Semester (CBCS) Examination

MODEL PAPER - I

PROGRAMMING IN C++

Time: 3 Hours Max. Marks: 80

SECTION - A (8Q × 4M = 32)

Answer any EIGHT questions. All questions carry equal marks.

1.	What is C++ ?	**(Unit - I, SQA. 1)**
2.	What are called as Inline Functions?	**(Unit - I, SQA. 7)**
3.	Overloading of a function.	**(Unit - I, SQA. 6)**
4.	Benefits of Object Oriented Programming.	**(Unit - II, SQA. 1)**
5.	What is a Class ?	**(Unit - II, SQA. 3)**
6.	Explain the rules to be followed in operator overloading.	**(Unit - II, SQA. 14)**
7.	Define virtual function.	**(Unit - III, SQA. 11)**
8.	Unformatted Input / Output Functions.	**(Unit - III, SQA. 15)**
9.	Define Abstract Class?	**(Unit - III, SQA. 13)**
10.	What is an Exceptions?	**(Unit - IV, SQA. 1)**
11.	Define template instantiation?	**(Unit - IV, SQA. 6)**
12.	What is a template? Explain the need of a template?	**(Unit - IV, SQA. 4)**

SECTION - B (4Q × 12M = 48)

Answer ALL questions. All questions carry equal marks.

13. (a) Describe the various operators in C++. **(Unit - I, Q.No. 11)**

OR

(b) Give an algorithm and explain the concept of selection sort with an example program. **(Unit - I, Q.No. 40)**

14. (a) What is the Difference Between Procedure Oriented Programming (POP) & Object Oriented Programming (OOP). **(Unit - II, Q.No. 2)**

OR

(b) Explain briefly about memberwise assignment with an example. **(Unit - II, Q.No. 40)**

15. (a) Describe briefly about multiple inheritance with an example. **(Unit - III, Q.No. 6)**

OR

(b) What is a polymorphism? Explain different types of polymorphism with example program. **(Unit - III, Q.No. 22)**

16. (a) How to handle an exception in C++. **(Unit - IV, Q.No. 4)**

OR

(b) What is function template? How to define it? Write a C++ program to add two numbers using function template. **(Unit - IV, Q.No. 13, 14)**

FACULTY OF SCIENCE

B.Sc. I-Year II-Semester (CBCS) Examination

MODEL PAPER - II

PROGRAMMING IN C++

Time: 3 Hours Max. Marks: 80

SECTION - A (8Q × 4M = 32)

Answer any EIGHT questions. All questions carry equal marks.

1. What is reference variable in C++. **(Unit - I, SQA. 9)**
2. Explain briefly about Control Structures. **(Unit - I, SQA. 10)**
3. Various operators in C++. **(Unit - I, SQA. 3)**
4. What are private member functions? **(Unit - II, SQA. 10)**
5. Define overloading. **(Unit - II, SQA. 13)**
6. Applications of OOP Technology. **(Unit - II, SQA. 9)**
7. What is Inheritance. **(Unit - III, SQA. 1)**
8. Rules associated with virtual function. **(Unit - III, SQA. 12)**
9. What is polymorphism? **(Unit - III, SQA. 9)**
10. List out various built-in exceptions. **(Unit - IV, SQA. 2)**
11. What is re-throwing an exception? **(Unit - IV, SQA. 3)**
12. What is Class Template ? **(Unit - IV, SQA. 9)**

SECTION - B (4Q × 12M = 48)

Answer ALL questions. All questions carry equal marks.

13. (a) Explain briefly about Control Structures. Write a Program to find whether the given number is divisible by 5 or 8. **(Unit - I, Q.No. 14, 19)**

OR

(b) Give an algorithm and explain the concept of linear or sequential search along with an example program. **(Unit - I, Q.No. 38)**

14. (a) Explain how to overload I/O operators with an example program. **(Unit - II, Q.No. 50)**

OR

(b) What is a Class ? Explain with an example. **(Unit - II, Q.No. 7)**

15. (a) Explain in detail various non-formatted (or) unformatted I/O functions. **(Unit - III, Q.No. 31)**

OR

(b) Define function redefining?Write a program to demonstration function redefining. **(Unit - III, Q.No. 20, 21)**

16. (a) Define Overloading with Template Function? Explain with an example program. **(Unit - IV, Q.No. 22)**

OR

(b) What is Class Template ? how to define it.Write a program to display Simple calculator using Class template **(Unit - IV, Q.No. 23, 26)**