LECTURE NOTES

UNIT- I

Object-Oriented Thinking : Different paradigms for problem solving, need for OOP paradigm, differences between OOP and Procedure oriented programming, Overview of OOP concepts - Abstraction, Encapsulation, Inheritance and Polymorphism.

C++ Basics: Structure of a C++ program, Data types, Declaration of variables, Expressions, Operators, Operator Precedence, Evaluation of expressions, Type conversions, Pointers, Arrays, Pointers and Arrays, Strings, Structures, References. Flow control statement- if, switch, while, for, do, break, continue, goto statements. Functions - Scope of variables, Parameter passing, Default arguments, inline functions, Recursive functions, Pointers to functions. Dynamic memory allocation and de-allocation operators - new and delete, Pre-processor directives.

C++ is a general-purpose and multi-paradigm computer programming language.

The C++ programming language supports both object-oriented programming and generic programming.

The C++ programming language is used to create applications that will run on a wide variety of hardware platforms such as personal computers running Windows, Linux, UNIX, and Mac OS.

The C++ programming language was created by **Bjarne Stroustrup** in the year 1983, at Bell Laboratories, USA.

The C++ programming language is said to be the superset of C programming language.

The programs written in C programming language can run in the C++ compiler.

The C++ programming language is an extension of the C programming language.

1.1 Different paradigms of problem solving:

The programming paradigm is the way of writing computer programs. There are four programming paradigms and they are as follows.

- Monolithic programming paradigm
- Structured-oriented programming paradigm
- Procedural-oriented programming paradigm
- Object-oriented programming paradigm

Monolithic Programming Paradigm:

The Monolithic programming paradigm is the oldest. It has the following characteristics. It is also known as the imperative programming paradigm.

- In this programming paradigm, the whole program is written in a single block.
- We use the **goto** statement to jump from one statement to another statement.
- It uses all data as global data which leads to data insecurity.

- There are no flow control statements like if, switch, for, and while statements in this paradigm.
- There is no concept of data types.

An **example** of a Monolithic programming paradigm is **Assembly language**.

Structure-oriented Programming Paradigm :

The Structure-oriented programming paradigm is the advanced paradigm of the monolithic paradigm. It has the following characteristics.

- This paradigm introduces a modular programming concept where a larger program is divided into smaller modules.
- It provides the concept of code reusability.
- It is introduced with the concept of data types.
- It also provides flow control statements that provide more control to the user.
- In this paradigm, all the data is used as global data which leads to data insecurity.

Examples of a structured-oriented programming paradigm is **ALGOL, Pascal, PL/I and Ada**.

Procedure-oriented Programming Paradigm :

The procedure-oriented programming paradigm is the advanced paradigm of a structure-oriented paradigm. It has the following characteristics.

- This paradigm introduces a modular programming concept where a larger program is divided into smaller modules.
- It provides the concept of code reusability.
- It is introduced with the concept of data types.
- It also provides flow control statements that provide more control to the user.
- It follows all the concepts of structure-oriented programming paradigm but the data is defined as global data, and also local data to the individual modules.
- In this paradigm, functions may transform data from one form to another.

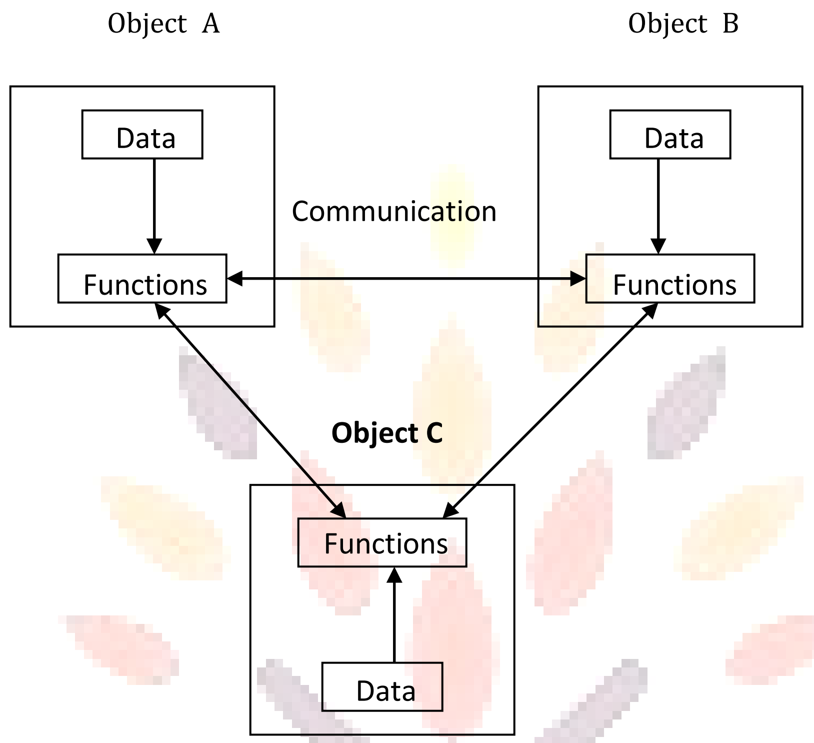
Examples of procedure-oriented programming paradigm is **C, visual basic, FORTRAN**, etc.

Object-oriented Programming Paradigm :

The object-oriented programming paradigm is the most popular. It has the following characteristics.

- In this paradigm, the whole program is created on the concept of objects.
- In this paradigm, objects may communicate with each other through function.
- This paradigm mainly focuses on data rather than functionality.
- In this paradigm, programs are divided into what are known as objects.
- It follows the bottom-up flow of execution.
- It introduces concepts like data abstraction, inheritance, and overloading of functions and operators overloading.
- In this paradigm, data is hidden and cannot be accessed by an external function.
- It has the concept of friend functions and virtual functions.

- In this paradigm, everything belongs to objects.



Examples of object-oriented programming paradigm is **C++, Java, C#, Python**, etc

1.2 Need for OOP Paradigm:

The structured programming made use of a top-down approach. To overcome the problems of structured programming, the object oriented programming concept was created.
The object oriented programming makes use of bottom-up approach. It also manages the increasing complexity.
The description of an object-oriented program can be given as, a data that controls access to code.
The object-oriented programming technique builds a program using the objects along with a set of well-defined interfaces to that object.
In OOP, data and the functionality are combined into a single entity called an object.
Classes as well as objects carry specific functionality in order to perform operations and to achieve the desired result.
The data and procedures are loosely coupled in procedural paradigm. Whereas in OOP paradigm, the data and methods are tightly coupled to form objects.
These objects help to build structure models of the problem domain and enable to get effective solutions.
OOP uses various principles (or) concepts such as abstraction, inheritance, encapsulation and polymorphism.
With the help of abstraction, the implementation is hidden and the functionality is exposed.
Use of inheritance can eliminate redundant code in a program.
Encapsulation enables the data and methods to wrap into a single entity.

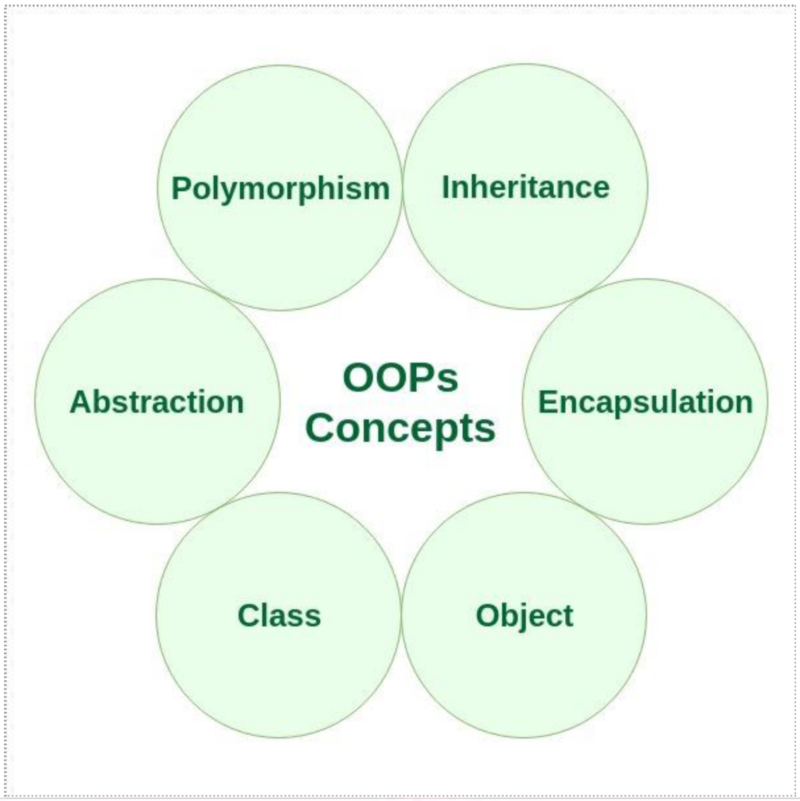
Polymorphism enables the reuse of both code and design.

1.3 Differences between POP and OOP:

POP	OOP
1. C was developed by Dennis Richie in 1972.	1. C++ was developed by Bjarne Stroustrup in 1979.
2. C is a structured / procedure oriented programming language.	2. C++ is an object oriented programming language.
3. C is a middle level language.	3. C++ is a high level language.
4. Importance is on procedure / steps to solve a problem.	4. Importance is on objects rather than procedure.
5. Functions are the fundamental building blocks in C.	5. Objects are the fundamental building blocks in C++.
6. C follows top-down approach.	6. C++ follows bottom-up approach.
7. C uses scanf() & printf() functions for standard input and output.	7. C++ use cin & cout streams for standard input & output.
8. In C, the program is divided into small parts called functions.	8. In C++, the program is divided into small parts called objects.
9. In C features like function overloading & operator overloading is not present.	9. C++supports function overloading & operator overloading.
10. In C control strings required.	10. In C++ control strings are notrequired.
11. C do not have any accessspecifiers.	11. C++ has access specifiers such as private, public, protected.
12. In C program file is saved with '.c' extension.	12. In C++ program file is saved with '**.cpp**' extension.

1.4 Overview of OOPs Concepts:

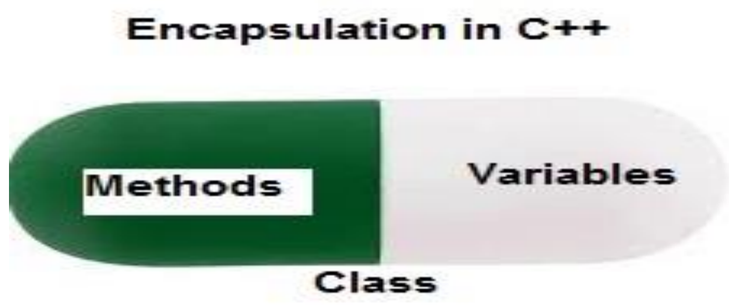
Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:



The following are basic concepts of OOP

1. Object
2. Class
3. Data Encapsulation
4. Data Abstraction
5. Inheritance
6. Polymorphism

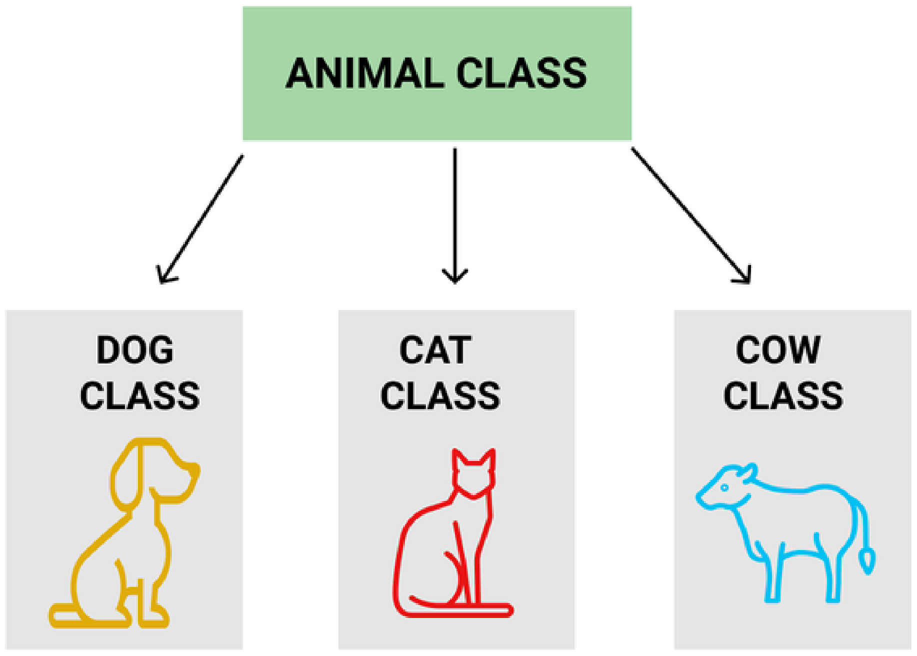
1. **Object:** It is a collection of no. of entities and they may represent a person or a place or a bank account ora table of data or any item that the program has to handle.
2. **Class:** It is similar to structures in C language. A class is a collection of objects of the same type.Class is a user defined data type in C++. Once a class has been defined we can create any number of objects belonging to that class.
3. **Data Encapsulation:** The process of combining data and functions into a single unit is called data encapsulation. With using encapsulation the data is not accessible outside the world only those functions which are present in the class can access the data.



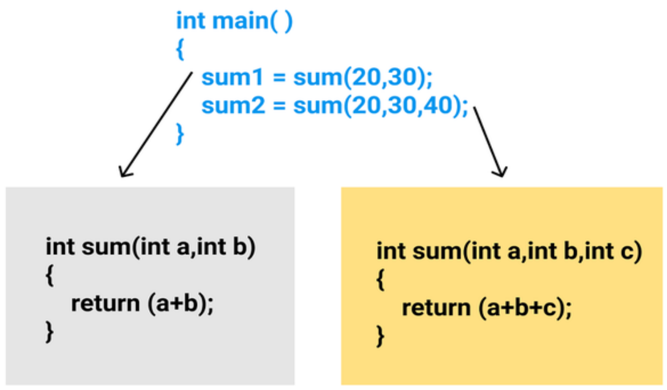
4. **Data Abstraction:** It refers to the act of representing essential features without including the background details or any explanations.

Inheritance: It is the process of objects of one class acquiring the properties of objects of another class. The main advantage of inheritance is to provide the data reusability i.e., we can add additional features tothe existing class without modifying it.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.



5. **Polymorphism:** It refers to the ability to take more than one form. In polymorphism an operation can exhibit different behaviors in different instances. For example consider the operation of addition of two numbers then operation will generate a sum. If the operands are strings then the operation wouldproduce a third string by concatenation.



```
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);
}
```

1.5 Structure of a C++ program:

Documentation section
Link section
Class declaration section
Member function definitions section

main()
{
Declaration part;
Executable part;
}

1. **Documentation section:** It is a set of comment lines that gives the complete information about theprogram.
Example: // Write a c program to find the simple interest//

2. **Link section:** It provides instructions to the compiler to link the functions from the system library i.e.,header files.
Examples:

#include<iostream.h>

# is the preprocessordirective command sign. include is a pre-processor directive command. iostream means input & output stream.

.h means extension of header file. #include<conio.h>

conio is a console input & output

3. <u>Class declaration section:</u>

class declaration is similar to structure declaration in C language. A class is a collection of objects of the same type. Class is a user defined data type in C++. Once a class has been defined we can create any number of objects belonging to that class. A class is a combination of data members and member functions.

<u>Syntax of class:</u>

class classname
{
Access specifier:
Data members/variable declarations;
Access specifier:
Member functions/function declarations;
};

In the above syntax the body of a class should be end with semicolon.

4. <u>Member function definitions section:</u>

A class is a combination of data members and member functions.This section declares member functions of a class.

Syntax:

returntype classname::memberfunction( )
{
Body of member function;
}

5. <u>main( )section:</u>

Every C++ program should contain one main() function and the C++ program executionstart with main( ). This section contains two parts.

1. Declaration part: This part declares all the variables which are used in the executable part.

Example:

```
int  a,b,c;
float x,y,z;
char a,b,c;
```

2. Executable part: This part contains a single statement or a group of statements.

All the statements in the declaration and executable parts should be end with semicolon (;).

1.6 Data types:

1. Data type indicates what kind of data can be stored in the variables or identifiers.
2. It can be used for storage representations.
3. Data types are fixed and pre defined meaning in C++ language.
4. Data types can be divided into four types: -

1.Primary / fundamental data types

Ex: - int,short int,long int,float,double,longdouble,char,bool,wchar_t
2.Derived data types
Ex: - arrays, pointers, functions, strings,structures,unions,reference
3.User defined data types
Ex: - typedef,enum,class.
4.Empty data type
Ex: - void (void data type is used for those function which does not return a value).

I.Primary/Fundamental data types: -
There are three types:-

1. Integer Data types
2. Character Data types
3. Real/Floating point Data types

1.Integer Data type: -

It can be used to represent/store integer constants.There are 3 types.
1. int 2.short int3.long int **1.int: -**

1. It is indicated by the keyword ‘int’.
2. It occupies of 2 bytes of memory location i.e., 16bits.

2. short int: -

1. It is indicated by the keyword ‘short’.
2. When the given value is less than the int range then short int is used.
3. It occupies 1byte of memory location i.e., 8bits.

3. long int: -

1.It is indicated by the keyword ‘long’.
2.When the given value is greater than the ‘int’ range then long int is used4.It occupies 4bytes of memory location i.e., 32bits.

2.Character Data types: -
It can be used to represent/store either single character constants or string character constants.

1.It is indicated by the keyword ‘char’.
3.It occupies 1byte of memory location i.e., 8bits.

3.Real/Floating point data types: -
It can be used to represent/store Real/Floating point constants.There are 3 types.
1.float 2.double 3.long double

4**1.float: -**

1) It is indicated by the keyword 'float'.
2) It occupies 4bytes of memory location. i.e., 32bits.

2.double: -

1) It is indicated by the keyword 'double'.
2) When the given value is greater than the float range then double is used.
3) It occupies 8bytes of memory location i.e., 64bits.

3.long double: -

1) When the given value is greater than the double range then long double is used.
2) It occupies 10 bytes of the memory location i.e., 80bits.

4.Additional datatypes in C++ :

1. **class:**
Class is a user defined data type in C++.
It is similar to structures in C language. A class is a collection of objects of the same type. Once a class has been defined we can create any number of objects belonging to that class. A class is a combination of data members and member functions.
Syntax of class: class classname

{

Access specifier:

Data members/variable declarations;Access specifier:

Member functions/function declarations;

};

2.boolean:
Boolean data type is used for storing boolean or logical values. A boolean variable can storeeither *true* or *false*. Keyword used for boolean data type is **bool**.

Syntax:
bool variablename;

3. wchar_t:
It is wide character literal constant. Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype. Represented by **wchar_t**. It is generally 2 or 4 bytes long.

4. **Reference:**
A reference is an alias (alternative name) for previously declared variable or object.

1.7 Declaration of variables :

Variables in C++ are a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In C++, all the variables must be declared before use.

A typical variable declaration is of the form:

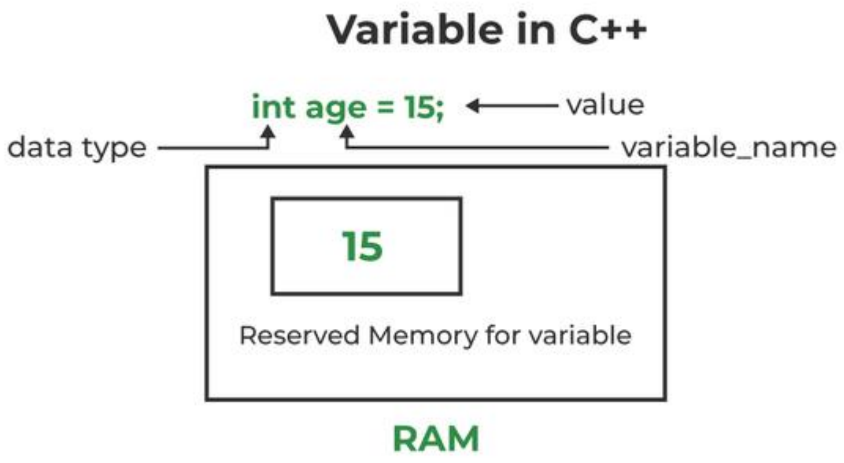
// Declaring a single variable

type variable_name;

// Declaring multiple variables:

type variable1_name, variable2_name, variable3_name;

A variable name can consist of alphabets (both upper and lower case), numbers, and the underscore '_' character. However, the name must not start with a number.



Initialization of a variable in C++

In the above diagram,

datatype**: Type of data that can be stored in this variable.**
variable_name**: Name given to the variable.**
value**: It is the initial value stored in the variable.**

Rules For Declaring Variable:

- The name of the variable contains letters, digits, and underscores.
- The name of the variable is case sensitive (ex Arr and arr both are different variables).
- The name of the variable does not contain any whitespace and special characters (ex #,$,%,*, etc).
- All the variable names must begin with a letter of the alphabet or an underscore(_).
- We cannot used C++ keyword(ex float,double,class)as a variable nam

1.8 Expressions:

C++ expression consists of operators, constants, and variables which are arranged according to the rules of the language. It can also contain function calls which return values. An expression can consist of one or more operands, zero or more operators to compute a value. Every expression produces some value which is assigned to the variable with the help of an assignment operator.

Examples of C++ expression:

1. (a+b) - c
2. (x/y) -z
3. 4a2 - 5b +c
4. (a+b) * (x+y)

1.9 Operators:

Operator is a symbol which specifies an operation to be performed between the operands.The following are the different operators available in C++.

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. sizeof() operator
5. Assignment operators
6. Conditional/Ternary operator
7. Increment and decrement operators
8. Bitwise operators
9. Specialoperators(Commaoperator(,),addressoperator(&),dotoperator(.),Valueata ddress operator(*),arrow operator(->))
10. Additional operators in C++

1.Arithmetic operator:-

1. Arithmetic operators are used to perform basic arithmetic operations between

theoperands.
2. There are 5 types of Arithmetic operators in C language they are +, -,* ,/ ,%.
3. Division operator(/) gives the result in quotient.
4. Modulus operator(%) gives the result in remainder.
5. % operator cannot be worked with the real constants.

2.Relational operators:-
1. Relational operators are used to compare the relation between the operands.
2. There are 6 types of relational operators in C language they are <, >, <=, >=,! = , ==.
3. Relational operator gives the result either in 1 (true) or 0 (false).

3.Logical operators:-
1. Logical operators are used to combine two or more relational conditions.
2. Logical operators gives the result either in 1 (true) or 0 (false).
3. There are 3 types of Logical operators in c language
    1. Logical AND (&&).
    2. Logical OR (||).
    3. Logical NOT (!).

4. **sizeof operator:-** This operator can be used to find the number of bytes occupied by avariable or data type.

Syntax:- sizeof(operand);

5. **Assignment operator:-**Values can be assigned to a variables using assignment operator in clanguage. The symbol of assignment operator is '='.
Assignment operator supports 5short hand assignment operators in C language they are += , - = , * = , /= , %=

6. Conditional operator/Ternary opertor:-
In C language '?' and ':' are called conditional operator or ternary operators in C language.
Syntax:- Exp1?Exp2:Exp3;

Here first Exp1 is evaluated, if Exp1 is true then Exp2 is executed otherwise Exp3 is executed.

7. Increment and Decrement operator:-
1. These operators are used to increment or decrement the value of the variable by 1.
2. ' ++ ' is the increment operator in C language.
3. '-- ' is the decrement operator in C language.

4.There are 4 types of operators in C language.

    1. Pre-increment operator
    2. Post-increment operator
    3. Pre-decrement operator
    4. Post-decrement operator

1. Pre-increment operator:

1. ++ operator is placed before the operand is called pre-increment operator. Ex: ++x,++a
2. Pre-increment operator specifies first increment the value of the variable by 1 and thenassigns the incremented value to other variable.

2.Post-increment operator:

1. ++ operator is placed after the operand is called post-increment operator. Ex: x++,a++
2. Post-increment operator specifies first assigns the value of the variable to other variableand then increments the value of the variable by1.

3. Pre-decrement operator:

1. -- operator is placed before the operand is called pre-decrement operator. Ex:--x,--a
2. Pre-decrement operator specifies first decrement the value of the variable by 1 and thenassigns the decremented value to other variable.

4. Post-decrement operator:

1. -- operator is placed after the operand is called post-decrement operator. Ex:x--,a--
2. Post-decrement operator specifies first assigns the value of the variable to othervariable and then decrements the value of the variable by1.

8.Bitwise operators: The data stored in a computer memory as a sequence of 0's and 1's.There are some operators works with 0's and 1's are called bitwise operators. Bitwise operatorscannot be worked with real constants.
There are 6 types of operators in C language.

1. Bitwise AND (&)
2. Bitwise OR (|)
3. Bitwise X-OR (^)
4. Bitwise left shift (<<)
5. Bitwise right shift (>>)
6. Bitwise 1's complement (~)

1.Bitwise AND (&): The result of the bitwise AND is 1 when both the bits are 1 otherwise 0.
2.Bitwise OR (|): The result of the bitwise OR is 0 when both the bits are 0 otherwise 1.
3.Bitwise X-OR (^): The result of the bitwise X-OR is 1 when one bit is 0 and other bit is 1otherwise 0.
4.Bitwise left shift (<<): This operator can be used to shift the bit positions to the left by 'n'positions.
5.Bitwise right shift (>>): This operator can be used to shift the bit positions to the right by 'n'positions.
6.Bitwise 1's complement (~): This operator can be used to reverse the bit i.e., it changesfrom 0 to 1 and 1 to 0.

9.Special operators-
Special operators(Comma operator(,),arrow operator(->),address operator(&),dotoperator(.),Value at address operator(*))

10.Additional operators in C++

1. Typecasting operator:
Converting one data type into another data type is called type casting operator in C++.Syntax:

(datatype)variable;

2.Insertion operator(<<):

This operator is used to print the output values on the monitor.

3.Extraction operator(>>):

This operator is used to read the data values from the keyboard.

4.Scope resolution operator(::):

It is to used to refer global data when the local and global variables have the same name.

Syntax:

```
returntype classname::memberfunction( )
{
Body of member function;
}
```

5.Memory management operators:

There are 2 types of operators in C++.

They are 1.new 2.delete

1. new operator:-

new operator can be used to create objects of any type .Hence new operator allocates sufficient memory tohold data of objects and it returns address of the allocated memory.

Syntax:

pointer-variable = **new** data-type;

Ex: int *p = new int;

2. delete operator:-

If the variable or object is no longer required or needed then it is destroyed by "**delete**" operator, there bysome amount of memory is released for future purpose.

Syntax:

delete pointer-variable;

Ex: delete p;

6.Manipulators:

These are used for formatting the data on the monitor.

Ex:endl and setw operators.

1.10 Operator Precedence :

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operators direction to be evaluated, it may be left to right or right to left.

Let's understand the precedence by the example given below:

```
int data=5+10*10;
```

The "data" variable will contain 105 because * (multiplicative operator) is evaluated before + (additive operator).

The precedence and associativity of C++ operators is given below:

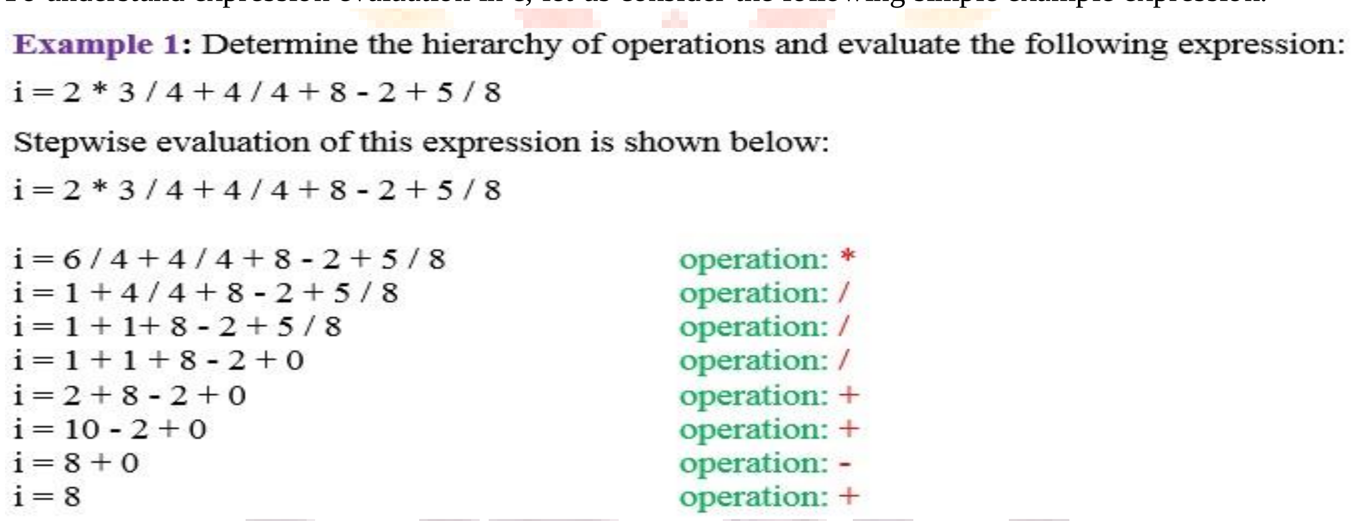
Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Right to left
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=/td>	Right to left
Bitwise AND	&	Left to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

Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

1.11 Expression Evaluation:

In the C++ programming language, an expression is evaluated based on the operator precedence and associativity. When there are multiple operators in an expression, they are evaluated according to their precedence and associativity. The operator with higher precedence is evaluated first and the operator with the least precedence is evaluated last.

To understand expression evaluation in c, let us consider the following simple example expression.



Example 1: Determine the hierarchy of operations and evaluate the following expression:

```
i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8
```

Stepwise evaluation of this expression is shown below:

```
i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8

i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8          operation: *
i = 1 + 4 / 4 + 8 - 2 + 5 / 8              operation: /
i = 1 + 1+ 8 - 2 + 5 / 8                   operation: /
i = 1 + 1 + 8 - 2 + 0                      operation: /
i = 2 + 8 - 2 + 0                          operation: +
i = 10 - 2 + 0                             operation: +
i = 8 + 0                                  operation: -
i = 8                                      operation: +
```

1.12 Type Conversion:

Type conversion is the process of converting a data value from one data type to another data type.

In the C++ programming language, the data conversion is performed in two different methods and they are as follows.

- Implicit Conversion (Type Conversion)
- Explicit Conversion (Type Casting)

Implicit Conversion (Type Conversion):

The type conversion is the process of converting a data value from one data type to another data type automatically by the compiler. Sometimes type conversion is also called **implicit type conversion**. The implicit type conversion is automatically performed by the compiler.

For example, in c programming language, when we assign an integer value to a float variable the integer value automatically gets converted to float value by adding decimal value 0. And when a float value is assigned to an integer variable the float value automatically gets converted to an integer value by removing the decimal value. To understand more about type conversion observe the following...

```
int     i   =  10   ;
float   x   =  15.5 ;
char    ch  =  'A'  ;
```

i = x ; =======> x value 15.5 is converted as 15 and assigned to variable i

x = i ; =======> Here i value 10 is converted as 10.000000 and assigned to variable x

i = ch ; =======> Here the ASCII value of A (65) is assigned to i

```
#include <iostream>
using namespace std;
int main()
{
   int i = 95 ;
   float f = 90.99 ;
   char ch = 'A' ;
   i = f ;    //float to int  --> 90.99 to 90
   cout << "i value is " << i << endl;
   f = i ;    // int to float --> 90 to 90.000000
   cout << "f value is " << f << endl;
   i = ch ;   // char to int  --> 'A' to 65
   cout << "i value is " << i << endl;

   return 0;
```

Explicit Conversion (Type Casting):

Typecasting is also called an **explicit type conversion**. Compiler converts data from one data type to another data type implicitly. When compiler converts implicitly, there may be a data loss. In such a case, we convert the data from one data type to another data type using explicit type conversion. To perform this we use the **unary cast operator**. To convert data from one type to another type we specify the target data type in parenthesis as a prefix to the data value that has to be converted. The general syntax of typecasting is as follows.

Example

int totalMarks = 450, maxMarks=600; ;
floataverage ;

average = (float) totalMarks / maxMarks * 100 ;

In the above example code, both totalMarks and maxMarks are integer data values. When we perform totalMarks / maxMarks the result is a float value, but the destination (average) datatype is a float. So we use type casting to convert totalMarks and maxMarks into float data type.

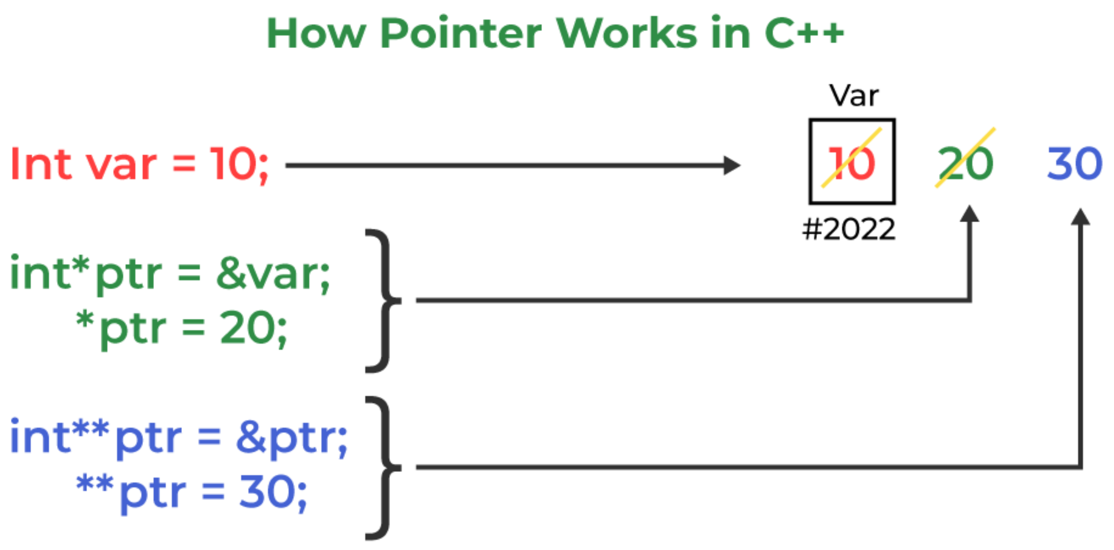
```
#include <iostream>

using namespace std;

int main()

{

   int a, b, c ;

   float avg ;

   cout << "Enter any three integer values : ";

   cin >> a >> b >> c;

   avg = (a + b + c) / 3 ;

   cout << "avg before type casting = " << avg << endl;

   avg = (float)(a + b + c) / 3 ;

   cout << "avg after type casting = " << avg << endl;

   return 0;
```

1.13 Pointers:

Pointers are symbolic representations of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. Iterating over elements in arrays or other data structures is one of the main use of pointers.

The address of the variable you're working with is assigned to the pointer variable that points to the same data type (such as an int or string).

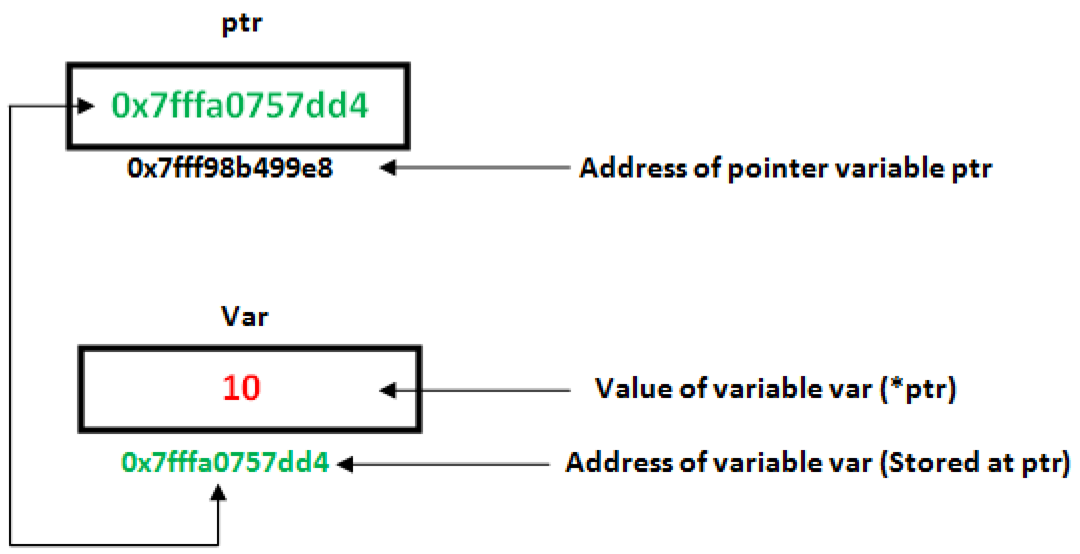
Syntax:
```
datatype *var_name;

int *ptr;   // ptr can point to an address which holds int data
```



How to use a pointer?

- Define a pointer variable
- Assigning the address of a variable to a pointer using the unary operator (&) which returns the address of that variable.
- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.

The reason we associate data type with a pointer is **that it knows how many bytes the data is stored in**. When we increment a pointer, we increase the pointer by the size of the data type to which it points.


ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (Stored at ptr)

```
// C++ program to illustrate Pointers
#include <bits/stdc++.h>
using namespace std;
void geeks()
{
    int var = 20;

    // declare pointer variable
    int* ptr;

    // note that data type of ptr and var must be same
    ptr = &var;

    // assign the address of a variable to a pointer
    cout << "Value at ptr = " << ptr << "\n";
    cout << "Value at var = " << var << "\n";
    cout << "Value at *ptr = " << *ptr << "\n";
}
// Driver program
int main()
{
  geeks();
  return 0;
}
```

Output

```
Value at ptr = 0x7ffe454c08cc

Value at var = 20
```

Value at *ptr = 20

1.14 Arrays :

- An array is a collection of data belonging to the same datatype and category, stored in contiguous memory locations.
- The size of the array remains fixed once declared. The indexing in the arrays always starts from 0.
- The memory locations in an array are contiguous, which means that the difference between adjacent addresses is equal to the size of elements belonging to that datatype.

Declaring an Array in C++:
The syntax for the declaration of C++ array:
datatype arrayname[size];
Eg: float a[10];

Initializing Arrays:
The common syntax for the initialization of a C++ array:
datatype arrayname[size]={element1,element2,.........,elementarraysize};

Example:

int a[5]={10,20,30,40,50};

1.15 Pointers and Arrays:

In C++, Pointers are variables that hold addresses of other variables. Not only can a pointer store the address of a single variable, it can also store the address of cells of an array.

Consider this example:

```
int  *ptr;
int  arr[5];
ptr=arr;
```

Here, ptr is a pointer variable while arr is an int array. The code ptr = arr; stores the address of the first element of the array in variable ptr.

Notice that we have used arr instead of &arr[0]. This is because both are the same. So, the code below is the same as the code above.

```
int  *ptr;
int  arr[5];
ptr=&arr[0];
```

The addresses for the rest of the array elements are given by &arr[1], &arr[2], &arr[3], and &arr[4].

Point to Every Array Elements

Suppose we need to point to the fourth element of the array using the same pointer ptr.

Here, if ptr points to the first element in the above example then ptr + 3 will point to the fourth element.

For example,

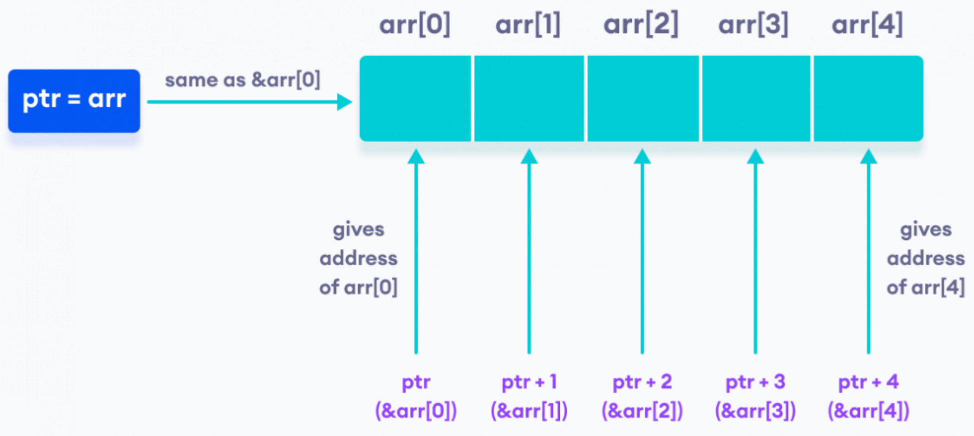
```
int  *ptr;
int  arr[5];
ptr=arr;
```

ptr+1 is equivalent to &arr[1];

ptr+2 is equivalent to &arr[2];

ptr+3 is equivalent to &arr[3];

ptr+4 is equivalent to &arr[4];



Working of C++ Pointers with Arrays

Note: The address between ptr and ptr + 1 differs by 4 bytes. It is because ptr is a pointer to an int data.

And, the size of int is 4 bytes in a 64-bit operating system.

Similarly, if pointer ptr is pointing to char type data, then the address between ptr and ptr + 1 is 1 byte. It is because the size of a character is 1 byte.

Example 1: C++ Pointers and Arrays

C++ Program to display address of each element of an array

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

int main()
{
    float arr[3];

    // declare pointer variable
    float *ptr;

    cout << "Displaying address using arrays: " << endl;

    // use for loop to print addresses of all array elements
    for (int i = 0; i < 3; ++i)
    {
        cout << "&arr[" << i << "] = " << &arr[i] << endl;
    }

    // ptr = &arr[0]
    ptr = arr;

    cout<<"\nDisplaying address using pointers: "<< endl;

    // use for loop to print addresses of all array elements
    // using pointer notation
    for (int i = 0; i < 3; ++i)
    {
        cout << "ptr + " << i << " = "<< ptr + i << endl;
    }

    return 0;
}
```

Output

```
Displaying address using arrays:

&arr[0] = 0x61fef0

&arr[1] = 0x61fef4

&arr[2] = 0x61fef8
```

```
Displaying address using pointers:

ptr + 0 = 0x61fef0

ptr + 1 = 0x61fef4

ptr + 2 = 0x61fef8
```

In the above program, we first simply printed the addresses of the array elements without using the pointer variable ptr.Then, we used the pointer ptr to point to the address of a[0], ptr + 1 to point to the address of a[1], and so on.

1.16 Strings :

Strings are used for storing text.

A string variable contains a collection of characters surrounded by double quotes.

Example:

Create a variable of type string and assign it a value:

```
string greeting = "Hello";
```

To use strings, you must include an additional header file in the source code, the <string> library:

Example:

```
// Include the string library
#include <string>

// Create a string variable
string greeting = "Hello";
```

String Concatenation

The + operator can be used between strings to add them together to make a new string. This is called **concatenation**:

Example:

```
string firstName = "John ";
string lastName = "Doe";
string fullName = firstName + lastName;
cout << fullName;
```

In the example above, we added a space after firstName to create a space between John and Doe on output. However, you could also add a space with quotes (" " or ' '):

Example:

```
string firstName = "John";
string lastName = "Doe";
string fullName = firstName + " " + lastName;
cout << fullName;
```

Append:

A string in C++ is actually an object, which contain functions that can perform certain operations on strings. For example, you can also concatenate strings with the append() function:

Example:

```
string firstName = "John ";
string lastName = "Doe";
string fullName = firstName.append(lastName);
cout << fullName;
```

1.17 Structure:

Structure is a collection of different data types. It is similar to the class that holds different types of data.

The Syntax Of Structure :

```
struct structure_name
{
   // member declarations.
}
```

In the above declaration, a structure is declared by preceding the **struct keyword** followed by the identifier(structure name). Inside the curly braces, we can declare the member variables of different types. **Consider the following situation:**

```
struct Student
{
  char name[20];
  int id;
  int age;
}
```

In the above case, Student is a structure contains three variables name, id, and age. When the structure is declared, no memory is allocated. When the variable of a structure is created, then the memory is allocated. Let's understand this scenario.

How to create the instance of Structure?

Structure variable can be defined as:

Student s;

Here, s is a structure variable of type **Student**. When the structure variable is created, the memory will be allocated. Student structure contains one char variable and two integer variable. Therefore, the memory for one char variable is 1 byte and two ints will be 2*4 = 8. The total memory occupied by the s variable is 9 byte.

How to access the variable of Structure:

The variable of the structure can be accessed by simply using the instance of the structure followed by the dot (.) operator and then the field of the structure.

For example:

1. s.id = 4;

In the above statement, we are accessing the id field of the structure Student by using the **dot(.)** operator and assigns the value 4 to the id field.

C++ Struct Example:

Let's see a simple example of struct Rectangle which has two data members width and height.

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

```
{
  int width, height;

 };
int main(void) {
    struct Rectangle rec;
    rec.width=8;
    rec.height=5;
  cout<<"Area of Rectangle is: "<<(rec.width * rec.height)<<endl;
 return 0;
}
```

Output:

```
Area of Rectangle is: 40
```

1.18 References in C++ :

When a variable is declared as a reference, it becomes an alternative name for an existing variable. A variable can be declared as a reference by putting '&' in the declaration.

Also, we can define a reference variable as a type of variable that can act as a reference to another variable. '&' is used for signifying the address of a variable or any memory. Variables associated with reference variables can be accessed either by its name or by the reference variable associated with it.

Syntax:

```
data_type &ref = variable;
```

C++ Program to demonstrate use of references :

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

int main()
{
    int x = 10;

    // ref is a reference to x.
    int& ref = x;

    // Value of x is now changed to 20
    ref = 20;
    cout << "x = " << x << '\n';
    // Value of x is now changed to 30
    x = 30;
    cout << "ref = " << ref << '\n';

    return 0;
}
```

Output:

```
x = 20
ref = 30
```

1.19 Flow control statements:

Control structures in C++ are of 3 types. They are

1. Conditional/Decision making control structures
2. Loop control structures
3. Unconditional control structures

1. Conditional/Decision making control execution statements.

There are 4 types of Conditional/Decision making control statements in C++ language. They are

1. if
2. if else
3. nested if else
4. switch statement

1. if statement: It can be used to execute a single statement or a group of statements based on the condition. It is a one way branching statement in C++ language.

Syntax:

```
if (condition)
{
Statements;
```

}

next statement;

Operation: First the condition is checked if it is true then the statements will be executed and then control is transferred to the next statement in the program.

if the condition is false, control skips the statements and then control is transferred to the next statement in the program.

2. **if else statement**: It can be used to execute a single statement or a group of statements based on the condition. It is a two way branching statement in C++ language.

Syntax:

if (condition)

{

statement1;

}

else

{

statement2;

}

next statement;

Operation: First the condition is checked if the condition is true then statement1 will be executed and it skips statement2 and then control is transferred to the next statement in the program.

if the condition is false it skips statement1 and statement2 will be executed and then control is transferred to the next statement in the program.

3. **nested if else statement**: If we want to check more than one condition then nested if else is used. It is multi way branching statement in C++ language.

Syntax:

if (condition1)

```
{
if (condition2)
{
statement1;
}
else
{
statement2;
}
}
else
{
statement3;
}
next statement;
```

Operation: First the condition1 is checked if it is false then statement3 will be executed and then control is transferred to the next statement in the program.

If the condition1 is true then condition2 is checked, if it is true then statement1 will be executed and then control is transferred to the next statement in the program.

If the condition2 is false then statement2 will be executed and then control is transferred to the next statement in the program.

4. **Switch statement**: If we want to select one statement from more number of statements then switch statement is used. It is a multi way branching statement in C++ language.

Syntax:

```
switch(expression)
```

```
{
case value1:block1; break;
case value2:block2; break;
--------
--------
case valuen:blockn; break;
default :default block; break;
}
next statement;
```

Operation: First the expression value (integer constant/character constant) is compared with all the case values in the switch. if it is matched with any case value then the particular case block will be executed and then control is transferred to the next statement in the program.

If the expression value is not matched with any case value then default block will be executed and then control is transferred to the next statement in the program.

2. **Loops/repetition control structures :**

There are 3 types of Loops/Repetition control statements in C++ language. They are

1. while
2. do while
3. for

1. **while loop statement:** It is used when a group of statements are executed repeatedly until the specified condition is true. It is also called entry controlled loop.The minimum number of execution takes place in while loop is 0.

Syntax:

```
while(condition)
{
```

body of while loop;

}

next statement;

Operation: First the condition is checked if the condition is true then control enters into the body of while loop to execute the statements repeatedly until the specified condition is true.

if the condition is false, the body of while loop is skipped and control comes out of the loop and continues with the next statement in the program.

2. **do while loop statement:-**It is used when a group of statements are executed repeatedly until the specified condition is true. It is also called exit controlled loop.

The minimum number of execution takes place in do while loop is 1. Syntax:-

do

{

body of do while loop;

}

while(condition); next statement;

In do-while loop condition should be end with semicolon.

Operation: In do-while loop the condition is checked at the end i.e., the control first enters into the body of do while loop to execute the statements. After the execution of statements it checks for the condition. if the condition is true then the control again enters into the body of do while loop to execute the statements repeatedly until the specified condition is true.

if the condition is false then control continues with the next statement in the program.

3. **for loop statement**:- It is used when a group of statements are executed repeatedly until the specified condition is true. It is also called entry controlled loop.The minimum number of execution takes place in for loop is 0.

Syntax:-

for (initialization;condition;increment/decrement)

{

body of for loop;

}

next statement;

Operation: First initial value will be assigned. Next condition is checked if the condition is true then control enters into the body of for loop to execute the statements. After the execution of statements the initial value will be incremented/decremented. After initial value will be incremented/decremented the control again checks for the condition. If the condition is true then the control is again enters into the body of for loop to execute the statements repeatedly until the specified condition is true.

if the condition is false, the body of for loop is skipped and control comes out of the loop and continues with the next statement in the program.

3. **Unconditional control structures:**

Normal flow of control can be transferred from one place to another place in the program. This can be done by using the following unconditional statements in the C++ language.

There are 3 types of Unconditional control statements in C++ language. They are

1. goto
2. break
3. continue

1. goto statement:- goto is an unconditional statement used to transfer the control from one statement to another statement in the program.

Syntax:- goto label; label:

Statements;

2. break statement:- break is an unconditional statement used to terminate the loops or switch statement. When it is used in loops(while,do while,for) control comes out of the loop and continues with the next statement in the program.

When it is used in switch statement to terminate the particular case block and then control is transferred to the next statement in the program.

Syntax:- statement; break;

3. continue statement:- continue is an unconditional statement used to control to be transferred to the beginning of the loop for the next iteration without executing the remaining statements in the program.

Syntax:- statement; continue;

1.20 Functions:

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function strcat() to concatenate two strings, function memcpy() to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure etc.

Defining a Function

The general form of a C++ function definition is as follows −

```
return_type function_name( parameter list ) {

   body of the function

}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

- Return Type − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

- Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

- Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- Function Body − The function body contains a collection of statements that define what the function does.

Example

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and return the biggest of both −

```
// function returning the max between two numbers

int max(int num1, int num2) {

   // local variable declaration

   int result;


   if (num1 > num2)

      result = num1;

   else

      result = num2;


   return result;

}
```

1.21 Scope of Variables in C++ :

In general, the scope is defined as the extent up to which something can be worked with. In programming also the scope of a variable is defined as the extent of the program code within which the variable can be accessed or declared or worked with. There are mainly two types of variable scopes:

1. Local Variables
2. Global Variables

Local Variables :

Variables defined within a function or block are said to be local to those functions.

•Anything between '{' and '}' is said to inside a block.

•Local variables do not exist outside the block in which they are declared, i.e. they can not be accessed or used outside that block.

•Declaring local variables: Local variables are declared inside a block.

```
// CPP program to illustrate  usage of local variables

#include<iostream>

using namespace std;


void func()

{

    // this variable is local to the

    // function func() and cannot be

    // accessed outside this function

    int age=18;

}
```

```
int main()

{

    cout<<"Age is: "<<age;



    return 0;

}
```

Output:

```
Error: age was not declared in this scope
```

The above program displays an error saying “age was not declared in this scope”. The variable age was declared within the function func() so it is local to that function and not visible to portion of p.

1.22 Parameter Passing Techniques in C/C++:

In C we can pass parameters in two different ways. These are call by value, and call by address, In C++, we can get another technique. This is called Call by reference. Let us see the effect of these, and how they work.

Call by value: In this technique, the parameters are copied to the function arguments. So if some modifications are done, that will update the copied value, not the actual value.

Example

```
#include <iostream>

using namespace std;

void my_swap(int x, int y) {

   int temp;

   temp = x;

   x = y;

   y = temp;
}

int main() {

   int a, b;

   a = 10;

   b = 40;

   cout << "(a,b) = (" << a << ", " << b << ")\n";

   my_swap(a, b);

   cout << "(a,b) = (" << a << ", " << b << ")\n";

}
```

Output

```
(a,b) = (10, 40)

(a,b) = (10, 40)
```

The call by address works by passing the address of the variables into the function. So when the function updates on the value pointed by that address, the actual value will be updated automatically.

Example:

```
#include <iostream>

using namespace std;

void my_swap(int *x, int *y) {

   int temp;

   temp = *x;

   *x = *y;

   *y = temp;

}

int main() {
  int a, b;

  a = 10;

  b = 40;

  cout << "(a,b) = (" << a << ", " << b << ")\n";

  my_swap(&a, &b);

  cout << "(a,b) = (" << a << ", " << b << ")\n";

}
```

Output:

```
(a,b) = (10, 40)

(a,b) = (40, 10)
```

Like the call by address, here we are using the call by reference. This is C++ only feature. We have to pass the reference variable of the argument, so for updating it, the actual value will be updated. Only at the function definition, we have to put & before variable name.

Example :

```
#include <iostream>

using namespace std;

void my_swap(int &x, int &y) {

  int temp;

  temp = x;

  x = y;

  y = temp;

}

int main() {
  int a, b;

  a = 10;

  b = 40;

  cout << "(a,b) = (" << a << ", " << b << ")\n";

  my_swap(a, b);

  cout << "(a,b) = (" << a << ", " << b << ")\n";

}
```

Output :

```
(a,b) = (10, 40)

(a,b) = (40, 10)
```

1.23 Default Arguments in C++ :

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn’t provide a value for the argument. In case any value is passed, the default value is overridden.

1) The following is a simple C++ example to demonstrate the use of default arguments. Here, we don’t have to write 3 sum functions; only one function works by using the default values for 3rd and 4th arguments.

```
// CPP Program to demonstrate Default Arguments

#include <iostream>

using namespace std;


// A function with default arguments,

// it can be called with

// 2 arguments or 3 arguments or 4 arguments.
```

```
int sum(int x, int y, int z = 0, int w = 0) //assigning default values to z,w as 0

{

    return (x + y + z + w);

}


// Driver Code

int main()

{

    // Statement 1

    cout << sum(10, 15) << endl;


    // Statement 2

    cout << sum(10, 15, 25) << endl;


    // Statement 3

    cout << sum(10, 15, 25, 30) << endl;

    return 0;

}
```

Output

```
25

50

80
```

1.24 Inline function:

C++ inline function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the inline specifier.

Following is an example, which makes use of inline function to return max of two numbers −

```
#include <iostream>

using namespace std;

inline int Max(int x, int y) {

   return (x > y)? x : y;

}


// Main function for the program

int main() {

   cout << "Max (20,10): " << Max(20,10) << endl;

   cout << "Max (0,200): " << Max(0,200) << endl;

   cout << "Max (100,1010): " << Max(100,1010) << endl;

   return 0;

}
```

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

```
Max (20,10): 20

Max (0,200): 200

Max (100,1010): 1010
```

1.25 Recursion:

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

The figure below shows how recursion works by calling itself over and over again.

How recursion works in C++ programming

The recursion continues until some condition is met.

To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and the other doesn't.

Factorial of a Number Using Recursion:

```
// Factorial of n = 1*2*3*...*n

#include <iostream>

using namespace std;

int factorial(int);

int main() {

    int n, result;

    cout << "Enter a non-negative number: ";

    cin >> n;

    result = factorial(n);

    cout << "Factorial of " << n << " = " << result;

    return 0;

    }
```

```
int factorial(int n) {

    if (n > 1)

    {

        return n * factorial(n - 1);

    }

    else

    {

        return 1;

    }

}
```

Output :

```
Enter a non-negative number: 4

Factorial of 4 = 24
```

1.26 Pointers and Functions :

As we know that pointers are used to point some variables; similarly, the function pointer is a pointer used to point functions. It is basically used to store the address of a function. We can call the function by using the function pointer, or we can also pass the pointer to another function as a parameter.

They are mainly useful for event-driven applications, callbacks, and even for storing the functions in arrays.

Syntax for Declaration of function pointer:

The following is the syntax for the declaration of a function pointer:

1. int (*FuncPtr) (int,int);

The above syntax is the function declaration. As functions are not simple as variables, but C++ is a type safe, so function pointers have return type and parameter list. In the above syntax, we first supply the return type, and then the name of the pointer, i.e., FuncPtr which is surrounded by the brackets and preceded by the pointer symbol, i.e., (*). After this, we have supplied the parameter list (int,int). The above function pointer can point to any function which takes two integer parameters and returns integer type value.

Calling a function indirectly

We can call the function with the help of a function pointer by simply using the name of the function pointer. The syntax of calling the function through the function pointer would be similar as we do the calling of the function normally.

Let's understand this scenario through an example.

```
#include <iostream>

using namespace std;

int add(int a , int b)

{

    return a+b;

}

int main()

{

 int (*funcptr)(int,int);  // function pointer declaration

 funcptr=add; // funcptr is pointing to the add function

 int sum=funcptr(5,5);

 std::cout << "value of sum is :" <<sum<< std::endl;

  return 0;

}
```

In the above program, we declare the function pointer, i.e., int (*funcptr)(int,int) and then we store the address of add() function in funcptr. This implies that funcptr contains the address of add() function. Now, we can call the add () function by using funcptr. The statement funcptr(5,5) calls the add() function, and the result of add() function gets stored in sum variable.

1.27 Dynamic memory allocation and de-allocation operators - new and delete

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on Heap and non-static and local variables get memory allocated on Stack .

The new operator is used to allocate a memory block, while the delete operator is used to de-allocate a memory block. In C language, we use malloc(), calloc(), and free functions, while in C++ language we use new and delete operators to allocate and de-allocate memory blocks at run-time.

Applications:

- One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.
- The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don’t need anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.

For normal variables like “int a”, “char str[10]”, etc, memory is automatically allocated and deallocated. For dynamically allocated memory like “int *p = new int[10]”, it is programmers responsibility to deallocate memory when no longer needed. If programmer doesn’t deallocate memory, it causes memory leak (memory is not deallocated until program terminates).

new operator:

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

- Syntax to use new operator: To allocate memory of any data type, the syntax is:
- pointer-variable = new data-type;

Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.

Example:

```
// Pointer initialized with NULL

// Then request memory for the variable

int *p = NULL;

p = new int;
```

OR

// Combine declaration of pointer

// and their assignment

int *p = new int;

- Initialize memory: We can also initialize the memory using new operator:
- pointer-variable = new data-type(value);

Example:

- int *p = new int(25);
- float *q = new float(75.25);
- Allocate block of memory: new operator is also used to allocate a block(an array) of memory of type data-type.
- pointer-variable = new data-type[size];

where size(a variable) specifies the number of elements in an array.

Example:

int *p = new int[10]

Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

delete Operator :

Memory de-allocation is also a part of this concept where the "clean-up" of space is done for variables or other data storage. It is the job of the programmer to de-allocate dynamically created space. For de-allocating dynamic memory, we use the delete operator.

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

Syntax:

```
// Release memory pointed by pointer-variable

delete pointer-variable;
```

Here, pointer-variable is the pointer that points to the data object created by new.

Examples:

```
  delete p;

  delete q;
```

To free the dynamically allocated array pointed by pointer-variable, use following form of delete:

```
// Release block of memory

// pointed by pointer-variable

delete[] pointer-variable;
```

Example:

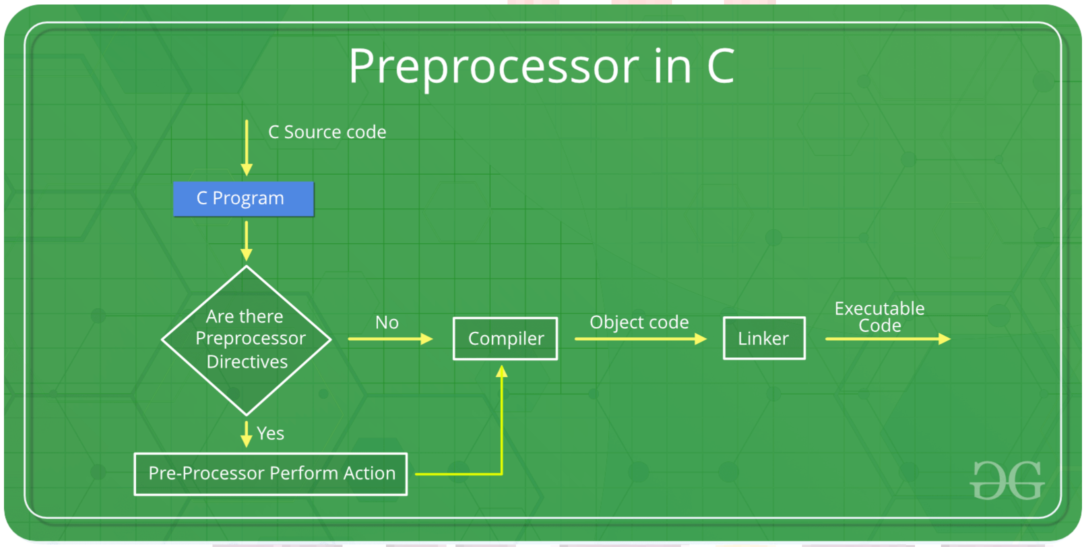
```
   // It will free the entire array

   // pointed by p.

   delete[] p;
```

Here's a simple program showing the concept of dynamic memory allocation:

```
#include <iostream>

using namespace std;

int main()

{

    double* val = NULL;

    val = new double;
    *val = 38184.26;

    cout << "Value is : " << *val << endl;

    delete val;

}
```

1.28 Preprocessor Directives:

Preprocessor programs provide preprocessor directives that tell the compiler to preprocess the source code before compiling. All of these preprocessor directives begin with a '#' (hash) symbol. The '#' symbol indicates that whatever statement starts with a '#' will go to the preprocessor program to get executed. We can place these preprocessor directives anywhere in our program.



Examples of some preprocessor directives are: #include, #define, #ifndef, etc.

Note: Remember that the # symbol only provides a path to the preprocessor, and a command such as include is processed by the preprocessor program. For example, #include will include the code or content of the specified file in your program.

These preprocessors can be classified based on the type of function they perform.

There are 4 Main Types of Preprocessor Directives:

1. Macros

2. File Inclusion
3. Conditional Compilation
4. Other directives

The following table lists all the preprocessor directives in C/C++:

Preprocessor Directives	Description
#define	Used to define a macro
#undef	Used to undefine a macro
#include	Used to include a file in the source code program
#ifdef	Used to include a section of code if a certain macro is defined by #define
#ifndef	Used to include a section of code if a certain macro is not defined by #define
#if	Check for the specified condition
#else	Alternate code that executes when #if fails

Preprocessor Directives	Description
#endif	Used to mark the end of #if, #ifdef, and #ifndef

UNIT-2

C++ Classes and Data Abstraction: Class definition, Class structure, Class objects, Class scope, this pointer, Friends to a class, Static class members, Constant member functions, Constructors and Destructors, Dynamic creation and destruction of objects, Data abstraction,ADT and information hiding.

C++ Classes and Data Abstraction:

2.1 Class Definition:

It is similar to structures in C language. A class is a collection of objects of the same type. Class is a user defined data type in C++. Once a class has been defined we can create any number of objects belonging to that class. A class is a combination of data members and member functions.

2.2 Class structure:

```
class classname
{
Access specifier:
Data members/variable declarations;
Access specifier:
Member functions/function declarations;
};
```

In the above syntax the body of a class should be end with semicolon. The class body contains the declaration of variables and functions. private and public are known as access specifiers in C++.private and public denote which of the members are private and which of the members are public. By default the members of a class are private.

Example:
Create a class called "MyClass":

```
class MyClass {       // The class
public:             // Access specifier
int myNum;        // Attribute (int variable)
string myString;  // Attribute (string variable)
};
```

- The class keyword is used to create a class called MyClass.
- The public keyword is an access specifier, which specifies that members (attributes and methods) of the class are accessible from outside the class. You will learn more about access specifiers later.
- Inside the class, there is an integer variable myNum and a string variable myString. When variables are declared within a class, they are called attributes.
- At last, end the class definition with a semicolon ;.

2.3 Class Objects:

A class object is a collection of no of entities and they may represent a person or a place or a bank account or a table of data or any item that the program has to handle. Object is an instance of a class.

Syntax :

classname objectname;

Example Program: Write a C++ program to find the addition of two numbers using class and object. #include<iostream>

```
using namespace std; class add

{

private:

int a,b; public:

int getdata(); int putdata();

};

int add::getdata()

{

cout<<"Enter the values of a,b\n"; cin>>a>>b;

}

int add::putdata()


{

int c; c=a+b;

cout<<"The addition of a,b is "<<c;

}
```

```
int main()

{

add x; x.getdata( );

x.putdata( );

}
```

2.4 Class Scope:

Class scope defines the accessibility or visibility of class variables or functions. The term scope is defined as a place where in variable name, function name and typedef are used inside a program. The different types of scopes are as follows,

1. Local scope
2. Function scope
3. File scope
4. Class scope and
5. Prototype scope.

1. Local Scope:

The variables that are declared in a function block are called local variable which have local scope. These variables can only be accessed by the function/block in which they are declared. The same variable can be used in other functions but it need to define their respective new scope. It is possible to insert a block within another block. The variable name given to a particular variable is local for the block and sub-block present inside it.

The formal parameter posses local scope and acts as it belongs to the outermost block.

Example:

```
void func

{

int i;

}
```

In the above function, since 'i' is declared inside the block, the integer 'i' has local scope. This integer can not be accessed because there is no code written for accessing it.

2. Function Scope:

The variables defined inside the function will have function scope. These variables can only be accessed inside the function.

Example:

```
void a()

{

//statements

int label;

{

goto label; //label in scope even though declared later

}

goto label; //label ignores block scope

}

void b( )

{

goto label;

}
```

3. File Scope:

The identifiers which are present outside the definition of function or class will have file scope. This scope consist of local scope as well as class scope and it will be a in the outermost part of program. The variables defined with file scope are known as global variables.

These variables can be accessed by all the functions in the program. Default value for global variable is zero.

Example :

```
int d; // Global variable declaration

void display( ); //Global function declaration
void main( )

{

d= 8; // Assignment of global variable 'd'

void display( ) // Global function called by main

{

cout<<"The value of d is" <<d<<"\n";

}

void display( )

{

cout<<d; //Global variable is accessible here also

}


}
```

4. Class Scope:

The data members and member functions defined in a class will have class scope. They are local only to that class. Each member possess unique scope.

Example:

```
class A

{

int a(int x = n)

{

//statements

return x * n;

}

int b( );
int i=n* 2;

static const int n = 1;

int A[n];

};

int A: : b()

{

return n;

}
```

5. Prototype Scope:

The variables that are declared inside a function prototype have prototype scope, They are local only up to that prototype.

Example :

```
int fun(int p, int q);

main ()

{

int a= 5, b = 10;

fun(a, b);

}

int fun(int p, int q)

{

int sum = p + q;

return sum;

}
```

2.5 this Pointer:

In C++ programming, this is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

- It can be used to pass current object as a parameter to another method.
- It can be used to refer current class instance variable.
- It can be used to declare indexers.

this Pointer Example :

Let's see the example of this keyword in C++ that refers to the fields of current class.

```
#include <iostream>

using namespace std;

class Employee {

   public:

    int id; //data member (also instance variable)

       string name; //data member(also instance variable)

       float salary;

       Employee(int id, string name, float salary)

        {

            this->id = id;

           this->name = name;

           this->salary = salary;

        }

       void display()

        {

           cout<<id<<"  "<<name<<"  "<<salary<<endl;

        }

};
```

```
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
    e1.display();
    e2.display();
    return 0;
}
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

2.6 Friends to a class :

The concept of encapsulation and data hiding is that a non-member function should not be allowed to access the private data members of that class.

In some situations, we would like to access the private data members of the class by using non-member functions or to access the data members of one object through another object.

In these situations C++ allows the common functions to be made friends with the classes, thereby allowing the friend functions to access the private data of these classes.

A friend function is a special function can access the private data of a class even though, it is not a member function of that class.

If a function is defined as a friend function then the private and protected data of a class can be accessed using that function.

The friend functions can be declared by using the keyword friend.

The function definition does not use either the keyword friend or scope resolution operator.

Syntax:

```
friend returntype functionname(classname objectname)
{
Body of a friend function;
}
```

Example Program

```
#include<iostream>
using namespace std;
class add
{
private:
int a,b;
public:
int getdata()
{
cout<<"enter the values of a,b\n";
cin>>a>>b;
}
friend int putdata(add s);
};
int putdata(add s)
{
int c;
c=s.a+s.b;
cout<<"The addition of a,b is "<<c;
}
int main()
{
add x;
x.getdata();
putdata(x);
}
```

2.7Static Data Members:

A data member of a class can be qualified as static. A static member variable has certain special characteristics:

1. It is initialized to zero when the first object of its class is created. No other initialization is permitted.

2. Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

3. It is visible only within the class, but its lifetime is the entire program.

4. Static data member is defined by keyword 'static'

Syntax:

datatype classname::static variablename;

Static Member Functions :

Like static member variable, we can also have static member functions. A member function that is declared static has the following properties:

A static function can have access to only other static members declared in the same class. A static member function is to be called using the class name as follows:

Syntax:

classname :: functionname;

Example Program

```
#include<iostream>
using namespace std;
class Demo
{
private:
```

```
static int x; static int y; public:

static int print( )

{

cout<<"Value of x= "<<x<<endl; cout<<"value of y= "<<y<<endl;

}

};

int Demo::x=10; int Demo::y=20; int main( )

{

Demo OB;

cout<<"printing through object name: "<<endl; OB.print();

cout<<"printing through class name: "<<endl; Demo::print();

}
```

2.9 Constructors and Destructors:

Constructors:

- A constructor is a special member function whose task is to initialize the objects of its class.
- It is special because its name is the same name as the class name.
- The constructor is invoked whenever an object of its associated class is created.
- It is called constructor because it constructs the values of data members of the class.
- A constructor does not have any return type.
- Constructors can be overloaded.
- Constructors should be declared in public section of a class.

Constructors are of 3 types:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

Default Constructor:

A constructor which has no arguments/parameters is called the default constructor.

Syntax:

```
 class    classname();
```

Example Program: Write a C++ program to implement default constructor.

```
 #include<iostream>
using namespace std;
class add
{
private:
int a,b; public:
add();
int calculate(); int display();
};
add::add()
{ a=10; b=20;
}
int add::calculate()
{
int c; c=a+b;
}
int add::display()
{
cout<<"Addition of two numbers= "<<c;
}
```

```
int main()

{

add x;

 x.calculate();

x.display();

}
```

Parameterized constructor:

A constructor which takes some arguments/parameters is called parameterized constructors.

Example Program: Write a C++ program to implement parameterized constructor.

```
 #include<iostream>

using namespace std; class add

{

private:

int a,b; public;

add(int x,int y);int calculate(); int display();

};

add::add(int x ,int y)

{

a=x; b=y;

}

int add::calculate()

{

int c; c=a+b;

}

int add::display()
```

```
{
cout<<" Addition of two numbers= "<<c;
}
int main()
{
add s(10,20);
s.calculate();
s.display();
}
```

Copy constructors:

It is used to copy the values of data members of one object into another object.

Example Program: Write a C++ program to implement copy constructor.

```
 #include<iostream>
using namespace std;
 class copyconst
{
private:
int a,b; public:
copyconst(int x,int y); int display();
};
copyconst::copyconst(int x,int y)
{
a=x; b=y;
}
int copyconst::display()
```

```
{
cout<<"a= "<<a<<"b= "<<b<<endl;
}
int main()
{
copyconst s1(10,20); copyconst s2=s1; s1.display();
s2.display();
}
```

Destructors:

- Destructor is a special member function which is called / invoked when the object is destroyed/deleted.
- Destructor is used to destroy the object as soon as the scope of object ends.
- Destructors should be same as class name but it is prefixed with ~ symbol.
- Destructors do not have any return type.
- Destructors cannot be overloaded.
- Destructor should be declared in public section of a class.

Syntax:

~classname();

2.10 Dynamic creation and destruction of objects:

The objects in C++ are dynamically created and destroyed by using new and delete operators.

(i) new Operator :

An object can be dynamically created by using a new operator that returns a pointer to it. A default constructor is called for the newly created object.

A special pointer called a NULL pointer is returned by the new operator if the requested memory is not available.

An object created by new is alive till delete is encountered.

General Form:

Pointer variable = new data type

Example:

int *ptr = new int;

In the above example, ptr is a pointer variable of type ‘int’ created by using a ‘new’ operator.

Pointer variable declaration along with the initialization can be done as,

int *ptr = new int(10);

Applications:

1. It can be used along with the classes and structures to allocate the memory.
2. It can also be used for creating multidimensional array, where the size of the error must be provided.

Advantages:

1. It finds the size of the object automatically without using the size operator.
2. Initializing the objects at the time of their declarations is allowed by the new operator.

Example Program:

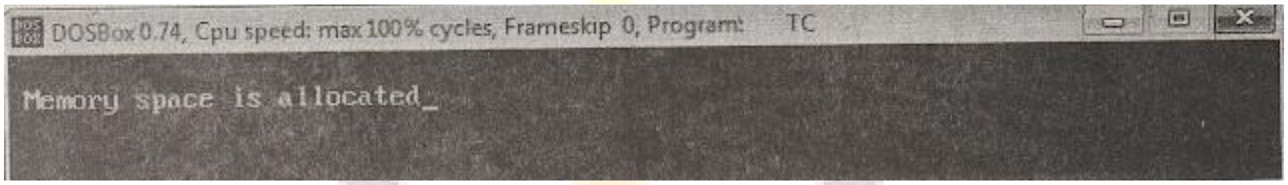
```
#include<iostream.h>

#include <conio.h>

int main()

{

int *p; //declaration of an integer

// pointer variable

clrscr( );

p=new int[15]; //allocation of memory for 15

//integer values

if(p=NULL)

cout<< "\n Memory space is released";

else

cout<< "\n Memory space is allocated”;
```

getch();

return 0;

}

Output


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

```
Memory space is allocated_
```

(ii) delete Operator :

This operator is used to deallocate the memory which was allocated by the new operator thereby destroying the object. The memory-is then released to the heap so that it can be utilized by some other object.

General Form

delete pointer variable;

Example:

delete P;

For array,

delete [size] P;

Where,

size = size of the array ‘P’. If no size is specified then the entire array 'P’ gets deleted.

When delete statement is encountered it calls the destructor apart. from deallocating the memory.

Example Program:

#include<iostream.h>

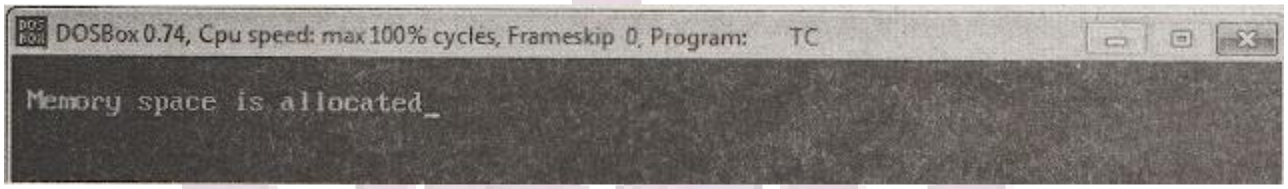
#include <conio.h>

int main()

{

```
 int *p; //declaration of an integer

// pointer variable :

clrscr( );

p=new int [15]; //allocation of memory for 15

//integer values

if(p==NULL)

cout<< "\n Memory space is released",

else

cout<< "\n Memory space is allocated";

delete p;//frees memory space allocation by new operator

getch();

return 0;

}
```

Output


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

```
Memory space is allocated_
```

2.11 Data Abstraction:

An abstraction is a process of showing relevant details and hiding irrelevant details from the user i.e., the implementation details are completely hidden from the user.

It gives the representation of real-world entities thereby showing only important attributes from the set of attributes.

To form groups from the set of attributes by using abstraction, then while forming the groups, the common attributes have been neglected hence, abstraction reduces the group complexity by simplifying its elements.

There are two types of abstraction. They are,

(i) Process abstraction

(ii) Data abstraction

(i) Process Abstraction:

It refers to the way in which the programs must specify some computational process has to be done. without providing the details of computation.

All function/subprograms and exceptional handlers are the examples of process abstraction.

(ii) Data Abstraction:

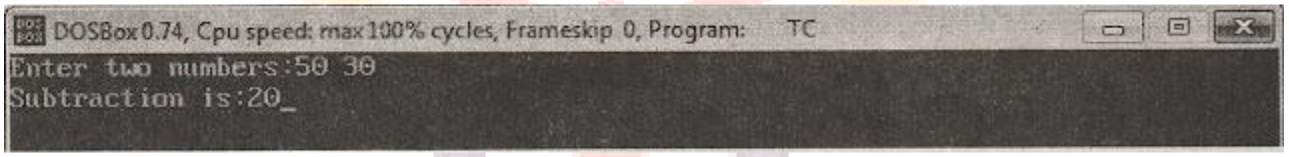
It refer to the way of isolating the data implementation methods from its usage. All the data members of an ADT are related to.data abstraction and the member functions to process abstraction. These two abstraction types are related. to each other.

Example Program:

```
#include<iostream.h>

#include<conio.h>

class Sub

{

private:

int x,y,z;

public:

void subtraction()

{

cout<<"Enter two numbers:";

cin>>x>>y;

z = x-y;

cout<<"Subtraction is:"<<z;

}

};
```

```
void main()
{
Sub s;
clrscr();
s.subtraction();
getch();
}
```

Output:



```
Enter two numbers:50 30
Subtraction is:20_
```

In the above program, ‘x’,‘y’ and ‘z’ are the private data-members and the function is the public member function. An object 's’ of type class is created in the main( ) function and then the member function subtraction( ) is called by using the dot operator.

2.12 Abstract Data Type (ADT):

Abstract Data Type (ADT) is a data type that allows the programmer to use it without concentrating on the details of its implementation.

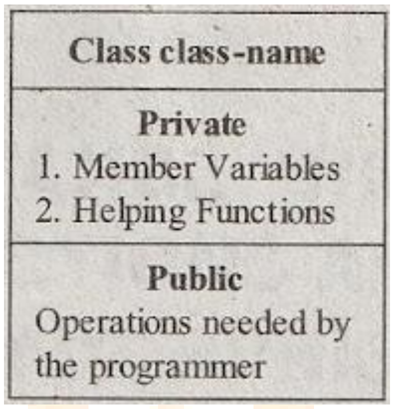
A class can be treated as an abstract data type by separating its specification from the implementation of its operations.

This separation between the specifications and the implementation can be obtained by,

(i) Considering and placing all the member variables in the private section of the class.

(ii) Placing all the needed operations in the public section and describing the ways of using these member functions.

(iii) Considering all the helping functions as private member functions and placing them in private section of the class.



Class class-name
Private 1. Member Variables 2. Helping Functions
Public Operations needed by the programmer

Table: Class as an ADT

The interface of an ADT specifies | how to use it in our program and consists of the public member functions along with the accompanying comments that describes the way to use these public member-functions.

The implementation of an ADT specifies the way to realize an interface as a C++ code and consists of private members of a class and the definitions of both the private and public member functions.

To use an ADT in our program, the only thing we need to know is its interface, but not its implementation i.e., implementation is not required to write the main program.

2.13 Information hiding:

Information hiding means hiding the class member variables and member functions. The process of hiding data involves hiding a class by declaring the data members and member functions within a “private” section. Mostly, member variables are declared as private and member functions as public. The data members that are declared as private cannot be accessed outside the class. Thereby, security is provided to the data. These private members are accessed by using public member functions. Generally, private and protected are the keywords that are used to provide security. The data that is encapsulated can also be called as ADT (Abstract Data Types). Information hiding is a process to build objects.

Example Program:

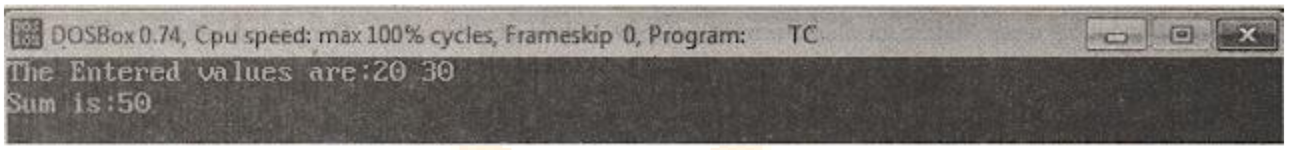
```
//Program to demonstrate information hiding is as given below,

#include<iostream.h>

#include<conio.h>

class Student
```

```
{
private:
int val1,val2;
int result;
public:
void calculate()
{
vall=20; .
val2=30;
result = vall+val2;
}
void display()
{
cout<<"The Entered values are:";
cout<<vall<<" "<<val2;
cout<<"\nSum is:"<<result;
}
};
int main()
{
Student stu;
clrscr();
stu.calculate();
stu.display();
getch();
```

return 0;

}

Output



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

```
The Entered values are:20 30
Sum is:50
```

<u>UNIT - III</u>

Inheritance: Defining a class hierarchy, Different forms of inheritance, Defining the Base and Derived classes, Access to the base class members, Base and Derived class construction, Destructors, Virtual base class.

Virtual Functions and Polymorphism: Static and Dynamic binding, virtual functions, Dynamic binding through virtual functions, Virtual function call mechanism, Pure virtual functions, Abstract classes, Implications of polymorphic use of classes,Virtual destructors.

<u>3. Inheritance:</u>

1. Inheritance is the process by which the objects of one class can acquires the properties of another class.
2. The mechanism of deriving a new class from an existing class is known as Inheritance.
3. The old class is called the super class/base class/parent class.
4. The new class is called the sub class/derived class/child class.
5. The derived class can inherits/acquires the properties/attributes from the base class.

<u>3.1 Defining a class hierarchy:</u>

In C++, it is possible to organize classes in the form. of a structure that corresponds to hierarchical inheritance. All the child classes can inherit the properties of their parent classes. A parent class that does not have any direct instance is called as an abstract class. It is used in the creation of sub-classes.

Example:

Let, ‘Meher’ be a florist, but a florist is a more specific form of shopkeeper. Additionally, a shopkeeper is a human and a human is definitely a mammal. But, a mammal is an animal and animal is a material object.

All these categories along with their relationships can be represented using a graphical technique as shown in figure. Each category is regarded as a class.

Inheritance is nothing but a principle, according to which knowledge of a category or a class which is more general car also be applied to a category or a class which is more specific.

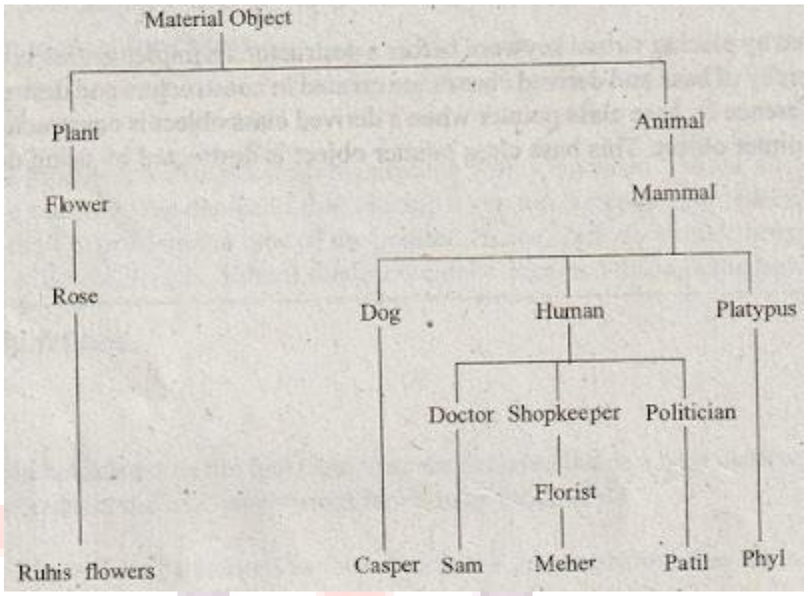


Figure: Class Hierarchy for Different Kinds of Material Objects

3.2 Different forms of Inheritance :

There are 5 types of inheritances in C++:

1.Single Inheritance

2.Multiple Inheritance

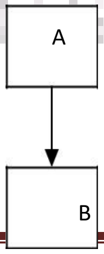
3.Multi Level Inheritance

4.Hierarchical Inheritance

5.Hybrid Inheritance

1.Single inheritance:

A single derived class can inherit the properties/attributes from single base class is called single inheritance.

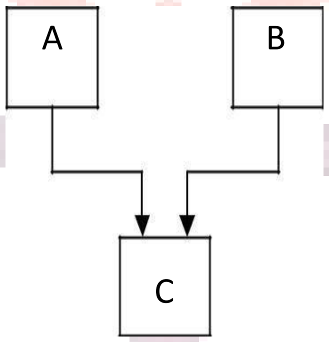


Syntax:

```
class baseclass
{
Body of base class;
};
class derived class:Accessspecifier baseclass
{
Body of derived class;
};
```

2.Multiple Inheritance:

In this type of inheritance a single derived class may inherit from two or more thantwo base classes.

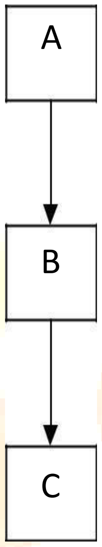


Syntax:

```
class baseclass1
{
Body of baseclass1;
};
class baseclass2
{
Body of baseclass2;
};
class derived class:Accessspecifier baseclass1,Accessspecifier baseclass2
{
Body of derived class;
};
```

3.Multi Level Inheritance:

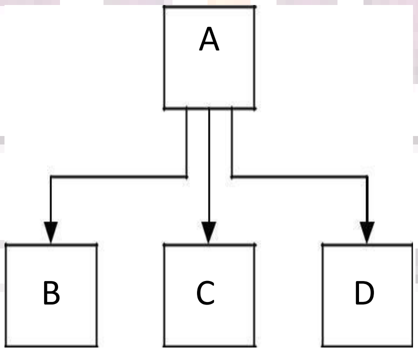
When a derived class is inherited from a base class (or) A base class itself inherited from a base class, thensuch inheritance is called multi level inheritance.



Syntax:
```
class baseclass
{
Body of baseclass;
};
class derivedclass1:Accessspecifier baseclass
{
Body of derived class 1;
};
class derived class2:Accessspecifier derivedclass1
{
Body of derived class 2;
};
```

4.Hierarchical Inheritance:

Deriving of multiple derived classes from a single base class is known as hierarchical inheritance.

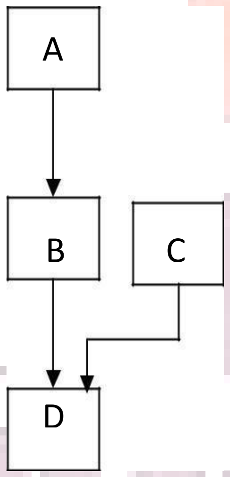


Syntax:

```
class baseclass
{
Body of baseclass;
};
class derivedclass1:Accessspecifier baseclass
{
Body of derived class 1;
};
class derivedclass2:Accessspecifier baseclass
{
Body of derived class 2;
};
```

5.Hybrid Inheritance:

Hybrid inheritance is combination of two or more inheritances such as single,multiple or multilevelinheritances.



Syntax:

```
class baseclass1
{
Body of baseclass1;
};
class derivedclass1:Accessspecifier baseclass1
{
Body of derivedclass1;
};
class baseclass2
{
Body of baseclass2;
};
class derivedclass2:Accessspecifier derivedclass1,Access specifier baseclass2
```

```
{
Body of derivedclass2;
};
```

3.3 Defining the Base and Derived classes :

Base Class:

A base class is a class in Object-Oriented Programming language, from which other classes are derived. The class which inherits the base class has all members of a base class as well as can also have some additional properties. The Base class members and member functions are inherited to Object of the derived class. A base class is also called parent class or superclass.

Derived Class:

A class that is created from an existing class. The derived class inherits all members and member functions of a base class. The derived class can have more functionality with respect to the Base class and can easily access the Base class. A Derived class is also called a child class or subclass.

Syntax for creating Derived Class:

```
class BaseClass{
  // members....
  // member function
}
class DerivedClass : public BaseClass{
  // members....
  // member function
}
```

3.4 Accessing of Base class members:

The base class members can be accessed by its sub-classes through access specifiers. There are three types of access specifies. They are public, private and protected.

1. Public:

When the base class is publicly inherited, the public members of the base class become the derived class public members.
They can be accessed by the objects of the derived class.

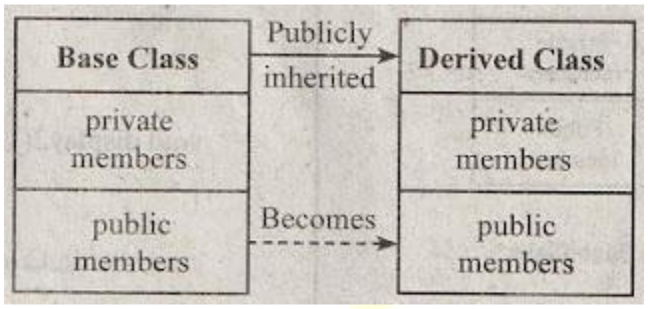


Figure: Publicly Inherited Base Class

Syntax:

class classname

{

public:

datatype variablename;

returntype functionname();

};

2. Protected:

When the base class is derived in protected mode, the ‘protected’ and ‘public’ members of the base class become the protected members of the derived class.
Private and protected members of a class can be accessed by,
(i) A friend function of the class.
(ii) A member function of the friend class.
(iii) A derived class member function.

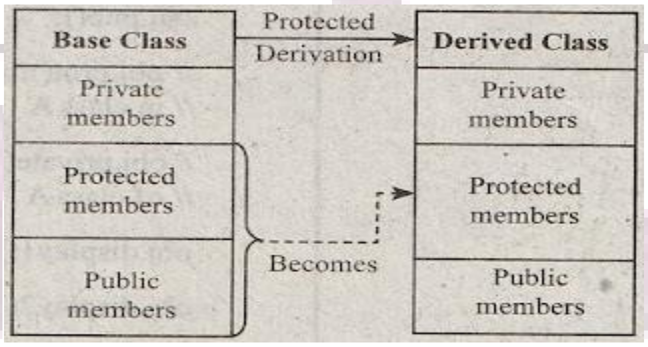


Figure: Protected Derivation of the Base Class

Syntax:

class classname

{

protected: :

datatype variablename;

returntype functionname();

};

3. Private :
When a base class contains members, that are declared as private, they cannot be accessed by the derived class objects. They can be accessed only by the class in which they are defined.

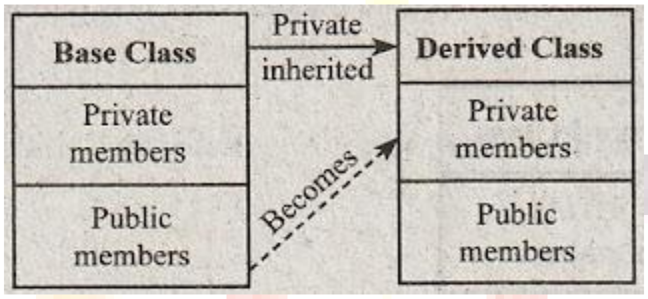


Figure: Privately inherited Base Class

Syntax:

```
class classname
{
private:
datatype variablename;
returntype functionname( );
};
```

Example Program:

```
#include <iostream>
using namespace std;
class A
{
public: pub()
{
cout << “In Public() \n”;
}
protected: prot()
{
cout << “In Protected() \n”;
```

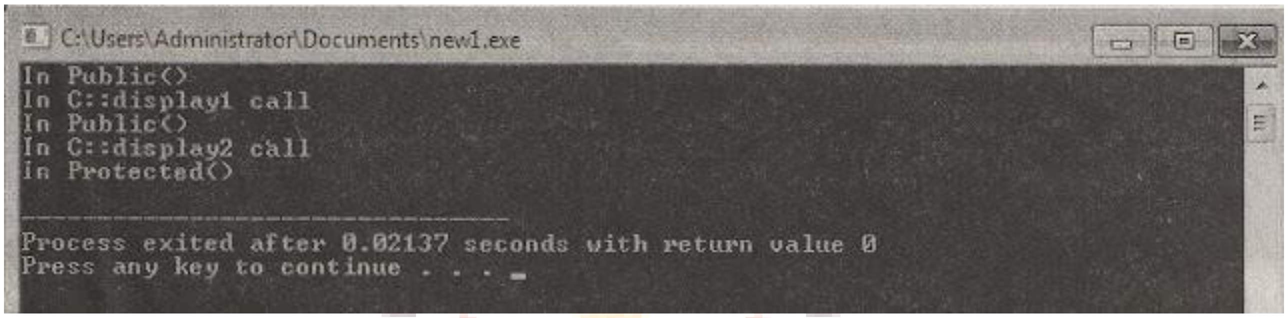
```
}
private: priv()
{
cout << "In Pivate() \n";
}
};
class C : public A
{
public:
void display1()
{
cout<< "In C::display1 call\n";
pub();
}
void display2()
{
cout << "In C::display2 call\n";
prot();
}
/*pri(Q is a private member of class A. Therefore it is an illegal access
void display3()
{
cout<<"In C::display3 call\n";
priv();
} */
}
main()
{
C obj;
obj.pub();
// obj.prot(); illegal because it is declared as protected in class A
// obj.private(); illegal because pri() isa private member of class A
```

obj.display1();

obj.display2();

}

Output


C:\Users\Administrator\Documents\new1.exe

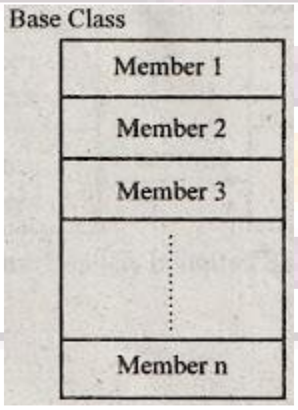
```
In Public()
In C::display1 call
In Public()
In C::display2 call
in Protected()

--------------------------------
Process exited after 0.02137 seconds with return value 0
Press any key to continue . . .
```

3.5 Base and derived class construction:

(i)Base Class:
Base class is a class from which other classes can be inherited. It is also called as a 'super class' or a 'parent class'. This class does not have any knowledge about its sub-classes. It is constant and cannot be changed. Any number of classes can be derived from a base class.
It is declared or defined before the derived class. The Base class members can be accessed from the derived class. It is used in creating the classes which reuse the code that is inherited from the base class. There are 2 types of base classes in C++.
They are as follows,

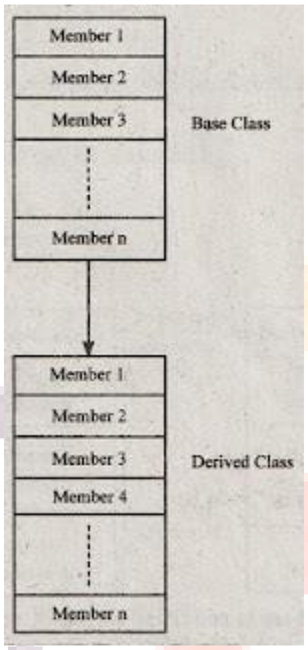
(a) Direct Base Class

(b) Indirect Base Class

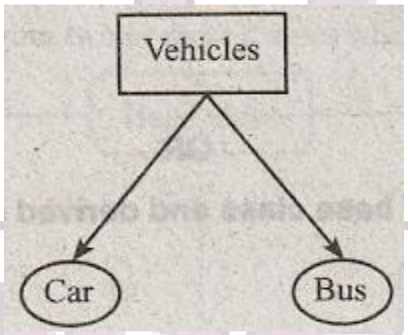


(ii) Derived Class :
Derived class is a class that is inherited from a base class. It can inherit all the properties and behavior of the base class. It contains additional members apart from having base class members.
It is also called as a 'subclass' or a 'child class'.
It has access to public and protected numbers of base class. It can define its own functions to

override the base class function. It can communicate with base class by calling the base class constructor.



Example :



In the above figure, vehicles is the Base class and the type of vehicles i.e., Car and Bus are derived class members. The derived classes car and Bus inherit the properties of Base class vehicle.

Syntax :

class Baseclass

{

};

class Derivedclass: AccessSpecifier Baseclass

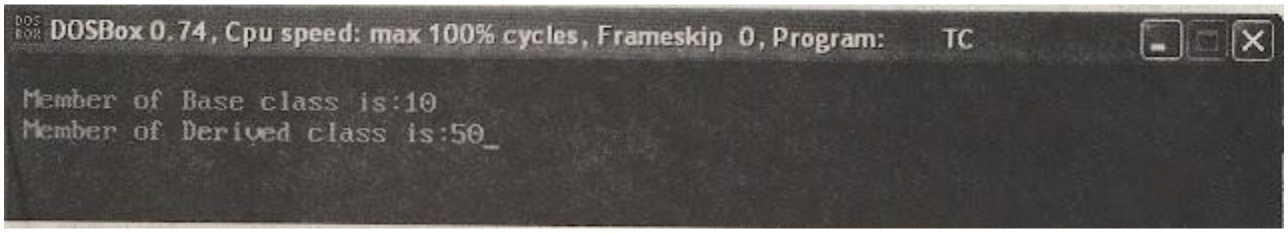
{

```
-------
};
```

Example Program:

```
#include<iostream.h>
#include<conio.h>
class Base
{
public:
int a;
};
class Derived : public Base
{
public:
int b;
};
int main()
{
clrscr();
Derived q;
q.a= 10;
q.b = 50;
cout<<"\n Member of Base class is:" <<q.a;
cout<<"\n Member of Derived class is:" <<q.b;
getch();
return 0;
}
```

Output:


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

```
Member of Base class is:10
Member of Derived class is:50_
```

3.6 Destructors:

A destructor is a special member function that is used to destroy the objects created by constructor. It takes the same name as the class but with a 'tilde' (~) at the beginning. It doesn't have any return type. It is called automatically at the end of the program execution to free up the acquired storage.
Memory allocation is done by using new operator in a constructor whereas delete operator is used in a destructor to deallocate the previously allocated memory.

3.7 Virtual Base class:

C++ has introduced the keyword 'virtual' to avoid the ambiguity that takes place by multipath inheritance. When the word virtual is used before the name of a class, it specifies that the class is virtual and indicates the compiler to take some essential caution in order to prohibit the duplication of member variables. Always a base class is declared as virtual, because it is the class from which other classes are derived. Thus, virtual base class is a class in which virtual keyword is placed before the name of base class while it is inherited.

3.8 Virtual functions and polymorphism:

3.8.1 Static and dynamic binding:

Static Binding:
Static binding or early binding is achieved through function overloading or operator overloading. Members with same name possessing different arguments are known to the compiler during compilation time. They help the compiler to find the function definition with appropriate function call. This type of binding is also called compile time binding.

Example Program:

```
class A //base class
{
inti;
public:
void display() //base class member function
{
```

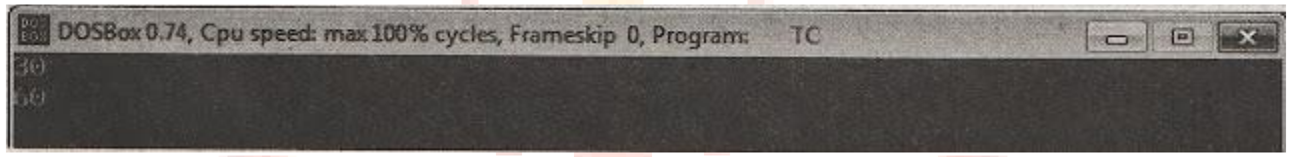
```
--------
--------
}
};
class B: public A // derived class.
{
int j:
public:
void display() // member function of derived class
{
--------
--------
}
};
```

In this example, both base class and derived class have same member function display( ) However, the addresses for these functions are different. When these functions are invoked, the control jumps to various addresses of those member functions. ‘

Example Program:

```
#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, Frameskip 0, Program:    TC
30
60
```

<u>Dynamic Binding :</u>

Dynamic binding or late binding binds a function definition to an appropriate function call during run time. This type of binding is also called run time binding. Dynamic binding of member functions is achieved using virtual keyword.

Example Program:

```
class A //base class
int i;
public:
virtual void display() //base class member function
{
--------
}
};
class B : public A //derived class
{
int j;
public:
virtual void display() //member function derived class
```

```
{
--------
}
};
```

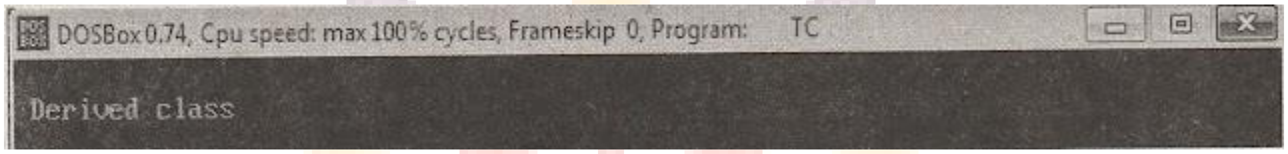
In this example, both base class and derived class has same member function display() declared using “virtual” keyword.
The virtual function must be defined in the public section. The system recognizes this and performs dynamic binding. It detects all the references from the base class and assumes that the virtual functions in derived class match with the base class functions (parameter and parameter type). If they do not match then functions are assumed as overloaded functions.

(ii) Runtime Polymorphism :
In this type of 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 late binding or dynamic binding. It is implemented using virtual functions and the pointers to objects.
Example Program:

```
#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, Frameskip 0, Program: TC

```
Derived class
```

3.8.2 Virtual functions :

A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.

It is used to tell the compiler to perform dynamic linkage or late binding on the function.

There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.

A 'virtual' is a keyword preceding the normal declaration of a function.

When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Rules of Virtual Function:

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.

- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor

Example of C++ virtual function used to invoked the derived class in a program.

```
#include <iostream>
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;    //pointer of base class
 B b;     //object of derived class
 a = &b;
 a->display();   //Late Binding occurs
}
```

Output:

```
Derived Class is invoked
```

3.8.3 Dynamic binding through virtual functions:

Dynamic binding in C++ can be implemented using virtual functions and polymorphism. Using virtual functions, we can declare a base class (parent class or superclass), and the derived classes can be used to override them.

```
#include <iostream>
using namespace std;
class base {
public:
   virtual void print() { cout << "print base class\n"; }

   void show() { cout << "show base class\n"; }
};

class derived : public base {
public:
   void print() { cout << "print derived class\n"; }

   void show() { cout << "show derived class\n"; }
};

int main()
{
   base* bptr;
   derived d;
   bptr = &d;

   // Virtual function, binded at runtime
   bptr->print();

   // Non-virtual function, binded at compile time
   bptr->show();

   return 0;
}
```

Output

```
print derived class

show base class
```

Explanation: Runtime polymorphism is achieved only through a pointer (or reference) of the base class type. Also, a base class pointer can point to the objects of the base class as well as to the objects of the derived class. In the above code, the base class pointer 'bptr' contains the address of object 'd' of the derived class.

Late binding (Runtime) is done in accordance with the content of the pointer (i.e. location pointed to by pointer) and Early binding (Compile-time) is done according to the type of pointer since the print() function is declared with the virtual keyword so it will be bound at runtime (output is *print derived class* as the pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time (output is *show base class* as the pointer is of base type).

Note: If we have created a virtual function in the base class and it is being overridden in the derived class then we don't need a virtual keyword in the derived class, functions are automatically considered virtual functions in the derived class.

3.8.4 Virtual functions call mechanism:

Virtual function call is usually implemented as an indirect function call through a per class table of function that are generated by compiler. Virtual function must be called by specifying objects pointed by the base pointers, so that it determines which function to be invoked. As the virtual function that is defined in base class is not required be to defined in derived class. Otherwise, the virtual function of base class will be used by default in all calls. If a virtual function is called through pointer or reference the actual object type is not known. For this reason, virtual function call mechanism is used.

Syntax:

```
class Base Class
{
public:
virtual void function name() // Virtual function definition
{
//statements
}
};
main()
{
Base_class pointer object;
pointer object → virtual function();  //Virtual function Call mechanism
}
```

In the above Syntax, virtual function is defined in base class. In the main function, the virtual function is called by using base class pointer.

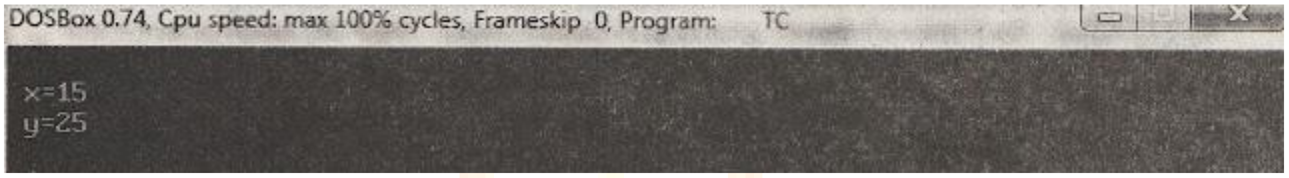
Example Program:

```
#include<iostream.h>
#include<conio.h>
class X
{
int x;
```

```
public: 2
X()
{
x=15;
}
virtual void display()
{
cout<<"\n x="<<x;
}
};
class Y: public X
{
int y;
public:
Y()
{
y = 25;
}
void display()
{
cout<<"\ny="<<y;
}
};
int main()
{
clrscr();
Y k;
b = &a;
b → display();
b = &k;
b → display();
getch();
```

```
return 0;
}
```

Output



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

```
x=15
y=25
```

3.8.5 Pure Virtual Function:

A pure virtual function (or abstract function) in C++ is a virtual function for which we can have an implementation, But we must override that function in the derived class, otherwise, the derived class will also become an abstract class. A pure virtual function is declared by assigning 0 in the declaration. They are also called as “do-nothing” functions as their definitions are empty and they are of the form,

```
virtual void display() = 0;
```

The display( ) in the above declaration is a pure virtual function with no definition relative to the base class. The assigned operator does not specify that zero is assigned to this function, instead it is used to tell the compiler that the declared function is a pure virtual function and that it will not have a definition.

Example of Pure Virtual Functions:

```
// An abstract class
class Test {
   // Data members of class
public:
   // Pure Virtual Function
   virtual void show() = 0;

   /* Other members */
};
```

Complete Example:

A pure virtual function is implemented by classes that are derived from an Abstract class.

```
// C++ Program to illustrate the abstract class and virtual functions
#include <iostream>
```

```
using namespace std;
class Base {
    // private member variable
    int x;
public:
    // pure virtual function
    virtual void fun() = 0;

    // getter function to access x
    int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived : public Base {
    // private member variable
    int y;
public:
    // implementation of the pure virtual function
    void fun() { cout << "fun() called"; }
};
int main(void)
{
    // creating an object of Derived class
    Derived d;
    // calling the fun() function of Derived class
    d.fun();
    return 0;
        }
```

Output:

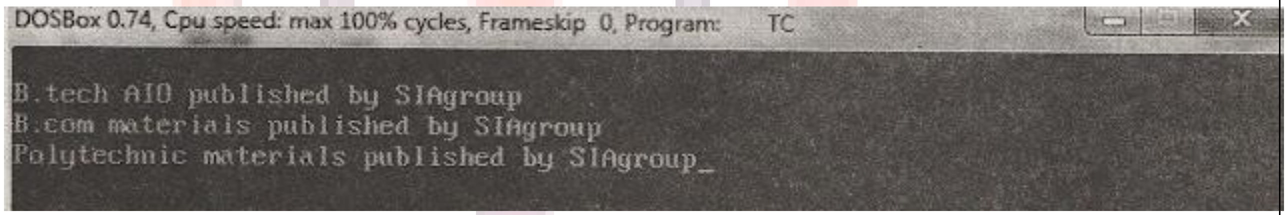
```
fun() called
```

Example Program:

```
#include<conio.h>
#include<iostream.h>
class SIAgroup
        {
        public:
        virtual void demo() = 0;
        };
class B-tech: public SIAgroup
{
public:
void demo()
        {
        cout<<“B.tech AIO published by SIAgroup”;
        }
};
class B.com: public SIAgroup
{
public
void demo()
        {
        cout<<“B.com materials published by, SIAgroup”;
        }
};
class Polytechnic: public SIAgroup
{
public
void demo():
        {
        cout<<“Polytechnic materials published by SIAgroup”;
        }
};
```

```
void main()
        {
        SlAgroup* array[3];
        B.tech sl;
        B.com s2;
        Polytechnic s3;
        array[0] = &s1;
        array[1] = &s2;
        array[2] = &s3;
        array[0] → demo();
        array[1] → demo();
        array[2] → demo();
        }
```

Output :


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

```
B.tech AIO published by SIAgroup
B.com materials published by SIAgroup
Polytechnic materials published by SIAgroup_
```

3.8.6 Abstract Classes:

Abstract classes are the classes that contain only function declaration, but not its definition. That is, it provides just the skeleton of the class hiding the implementation details. These classes are extendable and can even contain virtual functions, which help the programmer to debug the bugs. Abstract classes are defined in'a header file called abstract.h.

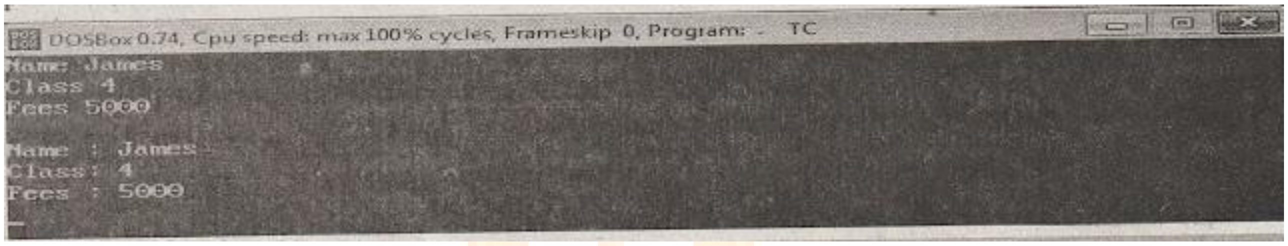
Example Program:

```
#include <iostream.h>
#include <conio.h>
const int max = 80;
class first
{
protected:
char name[max];
char cls[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 s1;
sl.insert();
sl.show();
```

getch();

}

Output :


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

```
Name: James
Class 4
Fees 5000

Name  : James
Class : 4
Fees  : 5000
```

3.8.7 Implications of polymorphic use of classes:

The programmer must be aware of some of the problems encountered while programming because the polymorphic use of class instances.will force to manipulate the objects either through pointers or references. Some of the problems are discussed as follows,

Declaring Destructor as Virtual :
Manipulation of objects through pointers require them to be created dynamically by using new operator and later on delete them through delete operator. Consider the below example,

```
class A
{
public A()
cout << “BaseClass Constructor” <endl;
}
~A()
{
cout << “Base Class Destructor” < endl;
};
class B: public A
{
public:
B()
{
cout << “Derived class constructor” << endl;
}
~B()
{
cout << "Derived class destructor" <<endl;
}
};
main()
{
A* a=new B;
delete a;
}
```

In the above example, a copy of derived class is created in main() by using the new operator and destroyed later on by using the delete operator. The output of this program is as shown below,

```
Baseclass constructor
Derived class constructor
Baseclass destructor.
```

The output is displayed based on the order in which the constructors are called.

The output does not show the destructor of derived class.This is because the derived class name is used with new operator to call the constructor, so the compiler created the specified object. The variable ‘a’ holds the pointer to base class. When this variable is defined with operator delete, the compiler could not predict that the programmer wants to destroy the derived class instance. This problem can be solved by declaring the destructor of base class as virtual. Now the destructor gets invoked through the virtual function table and the respective destructor will be called.

```
virtual ~A()
{
cout << "Baseclass Destructor" <<endl;
}
```

The output of the program when the above code is executed will be as shown below,

```
Base class constructor
Derived class constructor
Derived class destructor
Base class destructor.
```

Calling Virtual Functions in a Base class Constructor

The virtual functions may not work as expected some times. The undesired output of the virtual function call might result from the base class constructor. This problem is depicted by the below code,

```
class A
```

```
{
public:
A()
{
cout << “Base class constructor calls clone()” <<endl;
clone();
}
virtual void clone()
{
cout<< “Base class clone() called” <<endl;
}
};
class B: public A
{
public:
B()
{
cout << "Derived class constructor calls clone()" <<endl;
}
void clone()
{
cout << “Derived class clone() called” <<endl;
}
};
main()
{
Bb;
A*a=&b;
cout << “Call to clone() through instance pointer” <<endl;
a→clone();
}
```

The output of above code is as follows:

```
Baseclass constructor calls clone()
Base class clone() called
Derived class constructor calls clone()
Call to clone() through instance pointer
Derived class clone() called
```

As shown in the above output, the base class constructor is called first when a derived class instance is created. The derived class instance gets initialized partially. So, the compiler cannot bind the call to clone for derived class when virtual function clone() is called. Therefore, the compiler calls the clone() from base class.

The last two lines of output indicate that clone() function for derived class instance can be invoked using base class pointer upon object creation. This behavior is common in polymorphic use of virtual functions. The last line indicates that the compiler invokes base class version of function rather than that of derived class when virtual function is called in class constructor.

3.8.8 Virtual Destructor:

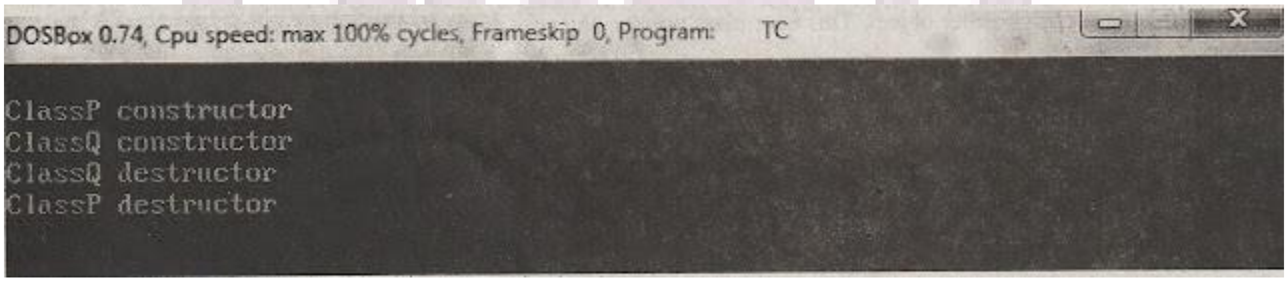
Virtual destructor is created by placing virtual keyword before a destructor. Its implementation is similar to the implementation of virtual constructors. The hierarchy of base and derived classes is created in constructors and destructors. From this hierarchy, derived and base class object reference by base class pointer when a derived class object is constructed using a new operator, It's address is stored in base class pointer object. This base class pointer object is destructed by using delete operator by invoking this operator.

Example Program:

```
#include<conio.h>
#include<iostream.h> .
class P
{
public:
P()
{
cout<<endl<<““ClassP constructor”;
}
virtual ~P()
{
```

```
cout<<endl<<ClassP destructor';
}
};
class Q : public P
{
public:
Q()
{
cout<<endl<<"ClassQ constructor";
}
~Q()
{
cout<<endl<<"ClassQ destructor";
}
};
void main()
{
clrscr();
P*ptr;
ptr=new Q;
delete ptr;
getch();
}
```

Output :


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

```
ClassP constructor
ClassQ constructor
ClassQ destructor
ClassP destructor
```

UNIT- IV

C++ I/O: I/O using C functions, Stream classes' hierarchy, Stream I/O, File streams and String streams, Overloading operators, Error handling during file operations, Formatted I/O.

4.C++ I/O :

Stream:

A stream can be defined as an interface between user and I/O devices. The I/O devices such as keyboard, disk, tape driver and monitor provides a source to perform I/O operations which include read and write operations. To perform I/O operations, I/O stream functions are mandatory and these functions are available in standard C++ library.
A standard library contains, .obj files and they are included in the program to perform various I/O operations. These operations provide portability to the program.
A stream can also be defined as flow of data. The flow can be calculated in terms of bytes in sequential manner. There are two types of streams, such as source stream and destination stream. The source stream receives data from the input devices such as keyboard and the data is considered as input data. Therefore, the other name for source Stream is given as input stream. On the otherhand, the destination stream collects data from the program and passes to output devices such as monitor. Therefore, the other name of destination stream is output stream.

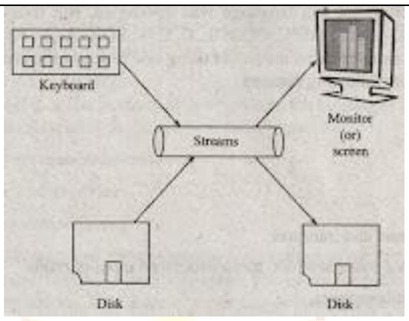


Figure: Block Diagram of 1/0 Devices and Streams

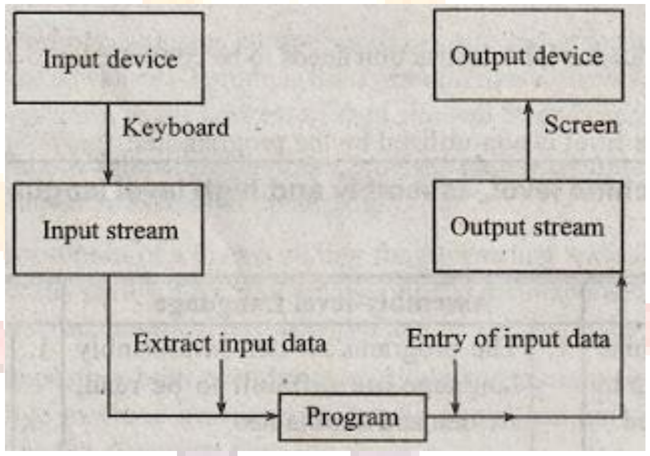


Figure: Block Diagram to Represent C++ Input and Output Streams

The input devices such as keyboard or'storage devices (hard disk, floppy disk) pass input data to input stream and output devices such as monitor or printer receive the output data from output stream.

4.1 I/O using C functions:

Formatted Input-output Functions
Input and output operations are performed using predefined library functions.
There are two formatted input/output functions.
(a) scanf(): Used for formatted input and
(b) printf(): Used to obtain formatted output
(a) scanf()
This function is used to read values for variables from keyboard.
Syntax

scanf("control string", address_list);

(i) Control String
Control string is enclosed within double quotes.
It specifies the type of values that have to be read from keyboard.
Control string consists of field specification written in the form.% field specification.
Field formats for different data types are given below,

Format	**Type of Value**
%d	Integer
%f	Floating Numbers

%c	Character
%ld	Long Integer
%u	Unsigned Integer
%lf	Double
%s	String type
%ox	Hexadecimal
%o	Octal
%i	Decimal or Hexadecimal or Octal
%e	Floating point number in exponential form
%g	Floating point number with trailing zeros truncated

(ii) Address_list

It contains addresses of input/output variables preceded. The addresses are specified by preceding a variable by '&' operator.

Examples

1. scanf("%d %f c", &i, &a, &b);

When user enters 10, 5.5, 'z' from keyboard, 10 is assigned to i, 5.5 is assigned to a and 'z' is assigned to b.

2. scanf ("%3d %2d", &a, &b);

If input is 500 and 10, then 500 is assigned to a and 10 is assigned to b

3. It is not always advisable to use field width specifiers in scanf statements. It may sometimes assign wrong values to variables as shown in example below.

scanf("%2d-%3d", &a, &b);

Here, if 5004 and 10 is input, then 50.is assigned to a and 04 to 6 which is wrong assignment. Hence, generally field widths are not used with scanf statement.

(b) printf()

printf() function is used to print result on-monitor.

Syntax

printf("control string", arg1, arg2,...., argn);

(i) Control string can have,

Format specifier given in the form % specifier

Escape sequence characters such as \t (tab), \n (new line), \b (blank) etc.

It can have a string that must be printed on console i.e., monitor.

(ii) arg1, arg2......, argn are variables whose values must be printed on the monitor in the format specified in control string.

Examples

1. printf("%d %c", num, ch);
2. printf('"hello world");
3. printf("%d \n %c", num, ch);

In example 3, suppose num has value 5 and ch has value 'a' then output will be printed as,

```
5
a
```

The output is printed in two different lines because new line character has been used between %d and %c.

4.2 stream class hierarchy:

Stream Classes:

Stream classes are set of classes, whose functionality depends on console file operations. The stream classes 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.

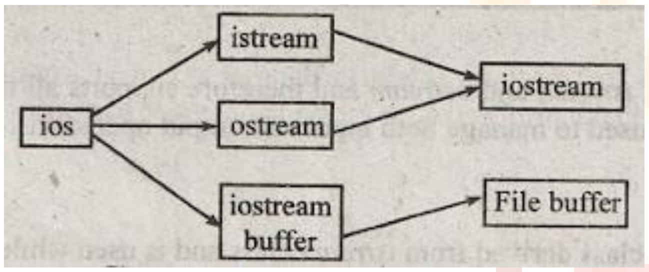


Figure (a): Stream Classes Hierarchy

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.

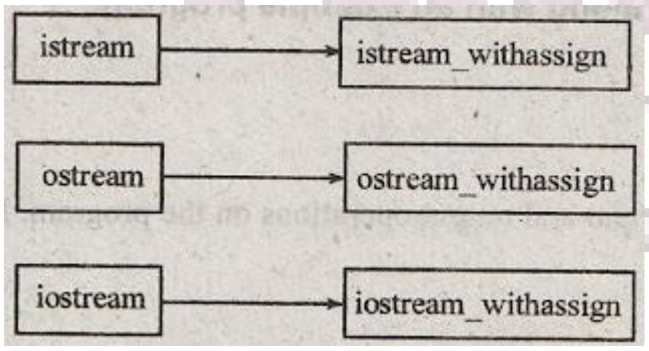


Figure (b): Other Stream Classes

Types of Stream Classes

The different stream classes include,

(a) iso

(b) istream

(c) ostream

(d) iostream

(e) istream_withassign

(f) ostream_withassign

(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 streambuf object. In addition to that, 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 a output stream class derived from ios class. This class handles the 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.

(e) 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.

4.3 Stream I/O:

I/O streams are used to perform input and output operations on the program. I/O streams include two types of streams.

They are as follows,

(a) Input stream

(b) Output stream.

(a) Input Stream

The input stream is used to read input through standard input device, keyboard. It uses a predefined stream object cin to perform the console read operation.

The cin object uses extraction operator (>>) to perform the input operation and this operator is stored in istream class.

Syntax

cin>> variable_name;

Example

char varl;

float var2;

int var3;

cin>> varl >> var2 >> var3;

Here varl, var2, var3 are different variables declared with different datatype. After the execution of declarative statements, memory is allocated for every variable. When cin statement is executed, input stream requests for the input. If the input is N 10.0 30, then the char value N is assigned to the variable var, the float value 10.0 is assigned to var2 and the int value 30 is assigned to var3.

If the given input is greater than the size of corresponding data type, the input remains in input stream. The data value of every variable is accepted through extraction operator >>. This operators reads the data values and stores the value in memory locations of that respective variable.

Cascading Input Operations

Cascading input operation is an input operation that allows more than one variable to read- input data. And, each variable in cin statement uses extraction operator. These variables must be separated with space, tab or enter.

(b) Output Streams

The output stream is used to control the output through standard output devices, monitor. It uses a predefined stream object cout to display the output. The cout object use insertion operator << to perform the console write operation and this operator is stored in ostream class.

Syntax

cout <<“output statement”;

cout << variable_name;

Example

cout <<“Hello world”;

cout<<varl, var2, ...;

Where, varl, var2, ... are variables.

In the above example, the first ‘cout’ statement displays the same text “Hello world” on to the screen. The second ‘cout’ statement displays the corresponding values of the variables varl, var2, ...

Program

```
#include <iostream.h>

#inelude <conio.h>

int main()

{

char stdId[10];

char stdName[20];

clrscr();

cout <<Enter student id” <<endl;

cin >> stdId;.

cout <<“Student ID is:” <<stdId;

cout <<endl; //breaks the line

cout <<“Enter student name:”;

cin >> stdName;

cout <<“Student name is:” << stdName;

getch();

return 0;
```

Output

```
Enter student id:SOGC0225

Student Id is : SOGC0225

Enter student name : Nymisha
```

Student name is : Nymisha

While reading string values, unwanted blanks spaces between characters are not allowed. In C++, format specifiers such as %f, %d, %u are not required.

4.4 File streams and String streams:

File streams:

Types of File Stream Classes

A file is a collection of related data stored in particular area on the disk.

The data transfer can take place in two ways,

1. Data transfer between the console unit and the program.

2. Data transfer between the program and a disk file.

File streams are used to transfer data between program and a disk file. Therefore a file stream acts as an interface between the programs and the files. The file stream I/O operations are similar to, consolestream I/O operations, The stream that provides the data to the program is called as an input stream and the stream that is used to receive the data from the program is called as an output stream. An input stream is used to extract the data from a file and an output stream is used to insert the data to a file. Performing input operations on file streams requires creation of an input stream and linking it with the program and the input file. Similarly performing output operations on file streams requires establishment of an output stream with the necessary links with the program and the output file.

The figure below shows the file stream hierarchy.

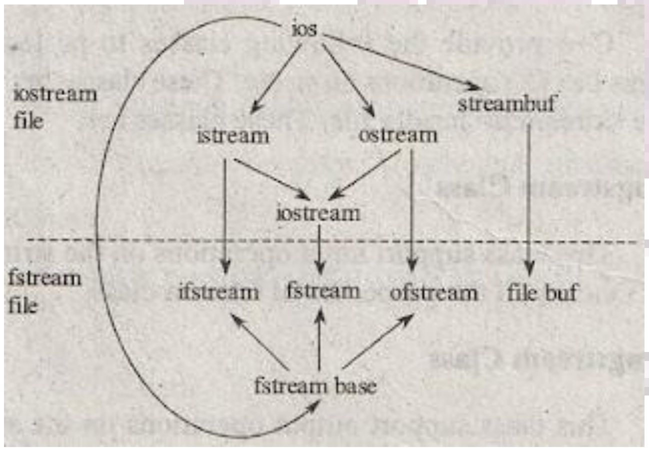


Figure: File Stream Class Hierarchy

C++ contains several classes that are used to perform file I/O operations. These are ifstream, ofstream and fstream classes which are derived from fstream base class and iostream class. The file stream classes are declared in the header file “fstream.h”.

1. ifstream Class

This class supports input operations on the files. It inherits the functions get( ), getline( ), read( ), seekg( ) and tellg( ) from istream class. When the file is opened in default mode, it contains open( ) function.

The functions of ifstream class are discussed below,

(i) get()

This function is used to read a single character from the file. The get pointer specifies the character which has to be read from the file. This function belongs to fstream class.

(ii) getline()

This function reads a line of text which ends with a newline character. It cam be called using cin object as follows,

cin.getline(line, size)

Where line is a variable that reads a line of text.

Size is number of characters to be read getline() function reads the input until it encounters ‘\n’ or size –1 characters are read. When ‘\n’ is read. it is not saved instead it is replaced by the null character.

(iii) read()

This function reads a block of data of length specified by an argument ‘n’. This function reads data sequentially. So when the EOF is reached before the whole block is read then the buffer will contain the elements read until EOF.

General syntax,

istream & read(char*str, size n);

(iv) seekg()

This function is used to shift the input pointer (i.e., get) to the given location. It belongs to ifstream. It is a file manipulator.

(v) tellg()

This function is used to determine the current position of the input pointer. It, belongs to ifstream class.

2. ofstream Class

This class supports output operations on the files. It inherits the functions put(), seekp(), tellp() and write() from ostream class. When the file is opened in default mode, it also contains the open() function.

The functions of ofstream class are discussed below,

(i) put();

This function is used to write a single character to the file. A stream object specifies to which file the character should be written. The character will be located at the position specified by the put pointer. This function belongs to fstream class.

(ii) . seekp()

This function is used to shift the output pointer (i.e., put) to the given location. It belongs to ofstream class. It is also a file manipulator.

(iii) tellp()

This function is used to determine the current position of the output pointer. It belongs to ofstream class.

(iv) write()

This function displays a line on the screen. It is called using cout object as follows, cout.write(line, size)

Where line is the string to be displayed.

Size is the number of characters to be displayed.

If the size of string is greater than the line (i.e., text to be displayed) then write() function stop displaying on encountering null character but displays beyond the bounds of line.

(3) fstream Class

This class provides support for both input and output operations on the files at the same time. It inherits all the member functions of the istream and ostream classes through iostream class. when a file is opened in default mode, it also contains open() function.

(4) fstream Baseclass

This is the base class for ifstream, ofstream and fstream classes. It provides the operations that are common to the file streams. It contains open( ) and class( ) functions.

(5) filebuf Class

This class is used to set the file buffers to read and write operations. it contains the functions open() and close(). It also contains a constant named Openport which is used

in opening of file stream classes using open() function.

(i) open()

This function is used to create new files as well as open existing files.

General Form

Stream-object open (“file name”, mode);

Where, ‘mode’ specifies the purpose of opening a file. This ‘mode’ argument is optional, if it is not given, then the prototype of member functions of the classes ifstream and ostream uses the default, values for this argument. The default value are,

ios :: in for ifstream functions i.e., open in read only mode

and ios :: out for ofstream functions i.e., open.in write only mode.

(ii) close()

A file can be closed using member function close(). This function neither takes any parameter nor returns any value.

Syntax

stream_name.close();

Here,

stream_name is the name of the stream to which file is linked.

String streams:

Hierarchy of string streams is as shown in the below figure,

String Streams

The figure below shows the hierarchy of string streams.

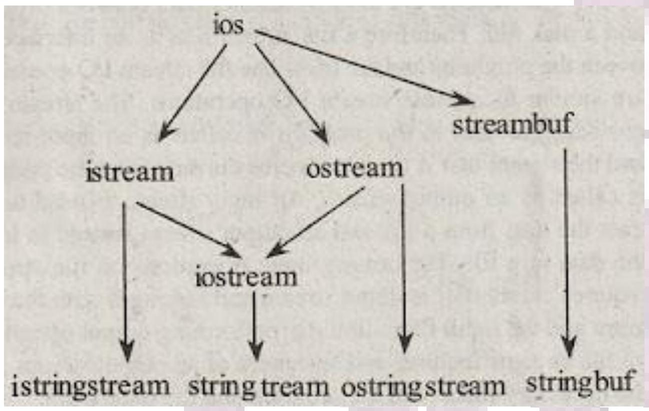


Figure: Hierarchy of String Streams

C++ provide the following classes to perform string streams i.e., I/O operations on string. These classes are declared in the <stream.h> header file. These classes are,

istringstream Class

This class support input operations on the strings. This class inherits all the properties of istream class.

ostringstream Class

This class support output operations on the strings. It inherits all the properties of ostream class.

stringstream Class

This class provides support for both the input and output operations on the strings at the same time. It is derived from the iostream class. So it inherits all the properties of iostream class.

stringbuf Class

This class is used to set the string buffers to read and 'write operations. That is this class provides virtual functions for I/O operations.

The following steps are required to read strings from stream,

1. First create the input string stream object of istringstream class.

2. Read the string passed as an argument to str() function into istring stream object.

The following steps are required to write strings to streams,

1. First create the output string stream object of ostream class.

2: Write the string from string stream using str() function.

For example, the following reads and writes strings into string stream.

```
#include<sstream.h>

int a, b;

string s = "44, 90";

istringstream iss; // Create input string stream

iss.str(s); // Specify string to read

iss>>a>>b; // Read strings.

ostring stream oss; // Declare output string stream

oss<<sqrt (b);

s = oss.str( ); // Get created string from output stream
```

Figure: Hierarchy of String Streams

C++ provide the following classes to perform string streams i.e., I/O operations on string. These classes are'declared in the <stream.h> header file. These classes are,

(i) istringstream class

(ii) ostringstream class

(iii) stringstream class

(iv) stringbuf class.

4.5 Overloading operators:

In C++, the operator << is called an insertion operator because it inserts the characters into a stream and the operator >> is called an extraction operator because it extracts characters from a stream. Like other operators such as +, —, *, etc. the operators >> and << can/also be overloaded. The functions that overload the insertion and extraction operators are called inserter and extractor respectively.
The prototypes definition for insertion (<<) and extraction (>>) operators are shown below.
Prototypes:

```
friend istream& operator>>(istream& streamobj, ClassName& classobj);
friend ostream& operator<<(ostream& streamobj, const ClassName& classobj);
```

Definitions

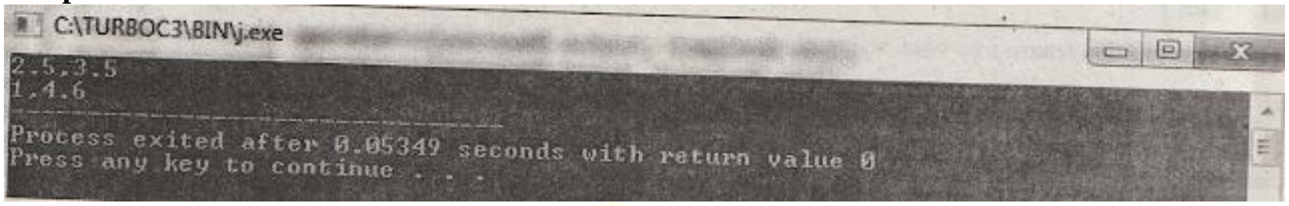
```
istreams& operator>>(istream& streamobj, ClassName& classobj)
{
// code
}
ostream& operator<<(ostream& streamobj, const ClassName& classobj)
{
/ code
}
```

Extraction Operator Overloading
The statement cin >> obj; is a statement that overloads the operator >>, like any other operator such as '+' operator, >> operator has two operands, one is cin i.e., the object of input stream and the other operand is an object of a class that receives the input value. Therefore the second argument passed to an overloaded operator >>( ) function must be a call by reference parameter. The purpose of this statement is to accept an input from the keyboard. When this statement is executed it calls the operator >>() function. The value returned by this function is an objegt of an input stream (istream).

Insertion Operator Overloading
The statement cout<<"Hello"; is an overloaded statement that overloads the operator <<. This operator also has two operands cout (object of output stream) and a string "Hello". The second operand may be either a string, a variable or a number.
The purpose of this statement is to print the string "Hello" to the screen. When this statement is executed it calls the overloaded operator >>() function. The value returned by this function is an object of the output stream (ostream).
Both the overloaded operator functions return a stream. The symbol '&' at the end of name of the stream represents that the operator function returns a reference i.e., it returns an object of the stream rather than returning the value of the stream itself.
The program below illustrates the operation of overloading both the insertion and extraction operators.
Example

```
#include <iostream>
using namespace std;
class Complex
{
private:
double real, imag;
public:
Complex() { }
Complex(double r, double i)
{
real=r;
imag =1;
}
friend ostream& operator<<(ostream& output, Complex& obj);
friend istream& operator>>(istream& input, Complex& obj);
};
ostream& operator<<(ostream& output, Complex& obj)
{
output<<obj.real<< ","<<obj imag;
return output;
}
istream& operator >>(istream& input, Complex& obj)
{
input >>obj.real>>obj.imag;
return input;
}
int main()
{
Complex obj1(2.5, 3.5);
Complex obj2(1.0, 4.6);
cout<<obj 1<<endl<<obj2;
}
```

Output



```
2.5,3.5
1,4.6
--------------------------------
Process exited after 0.05349 seconds with return value 0
Press any key to continue . . .
```

4.6 Error handling during file operations:

While performing input/output operations on the file one may encounter any of the following unknown conditions.

A file being opened doesn’t exist,

The file name chosen for a new file already exists.

The possibility of the disk being full.

Using a disk that is write protected.

Using an invalid name for a file.

An attempt to perform an operation when the file is not open for that purpose and so on.

Checking and handling all these unknown conditions is called error handling during file operations.

The C++ file stream uses a stream-state member of the is class. This member maintains the information on the status of a file that is currently being used. This member also uses bit fields to store the status of error conditions. The meaning of each bit of the stream-state is shown below,

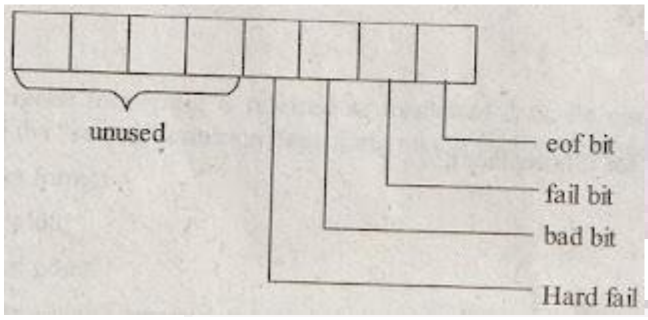


Figure: Stream-State Field

eof bit

This bit indicates having reached of end-of-file.

fail bit

This bit indicates that an operation failed. It may be due to formatting,

bad bit

This bit indicates some invalid Operation occurred or something is wrong with the buffer.

hard fail

This bit indicates irrecoverable error.

The function ios :: tdstate() is used to obtain the value of the stream-state variable.

The ios class provides several functions like eof(), good () fail() and bad( ) to read the status recorded ina file stream.

These functions are discussed below, .

(a) eof()

This function returns true (or non-zero value) on reaching end-of-file while reading i.e., if EOF flag is set otherwise it returns false (zero).

(b) fail()

This function returns true when an input/output operation has failed, i.e. if fail bit, bad bit or hard fail flag is set.

(c) bad()

If an invalid operation is performed or any irrecoverable error has ‘occurred then this function returns true ie., if bad bit or hardfail flag is set. However it returns false if it is possible to recover from any other reported error and continue the operation.

(d) good()

If no error has occurred then this function returns true. That is, all the above functions are false and I/O flag is set. And a program can proceed to perform I/O operations. If this function returns false then it means that the program can’t perform further Operations.

(e) clear()

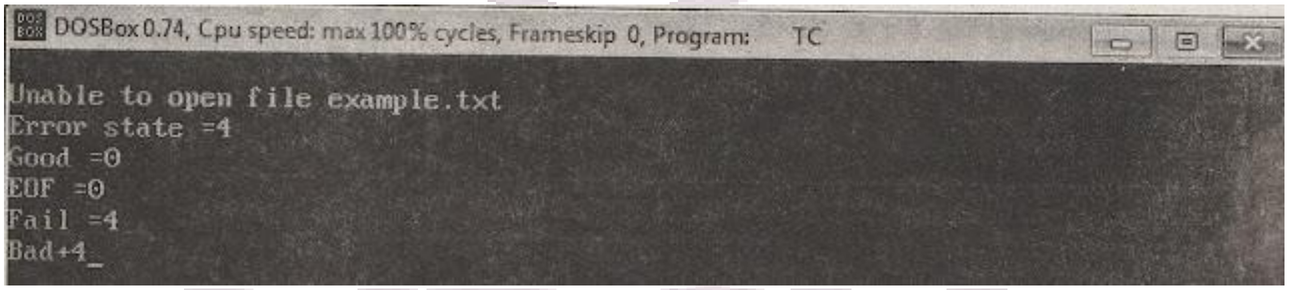
When this function is called with no arguments, it clears all error bits. This is also used to set specified flags. For example clear (ios :: failbit).

The following program illustrates the use of these functions. Assume that the file “Example.txt” already exists.

Program

```
#include<fstream.h>

#include<stdlib.h>

#include<conio.h>

void main()

{

void warning(ofstreame&);

ofstream str;

clrscr();

str.open("Example.txt", ios:: noreplace);

if(!str)

{

warning(str);

exit(1);

}

else

{

str<<"This text is written into the file example.txt'";

if(!str)

{

warning(str);

exit(2);

}

str.close();

}
void warning(ofstream &str)

{

cout << endl <<“Unable to open file example.txt”;

cout <<endl <<“Error state =” <<str.rdstate();

cout <<endl <<“Good =” <<str.good();

cout <<endI <<“EOF =” <<str.eof();

cout <<endl <<“Fail =” <<str.fail();

cout <<end] <<“Bad+” <<str.bad();

getch();

}
```

Output


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

```
Unable to open file example.txt
Error state =4
Good =0
EOF =0
Fail =4
Bad+4_
```

4.7 Formatted I/O:

Formatted Data

The data which has undergone formatting is referred as formatted data. Formatting is a way of representing data by performing necessary changes to the “setting” criterion depending on the user’s requirement. The various settings include,

(i) Changing the number format

(ii) Modifying the field width

(iii) Changing the decimal point.

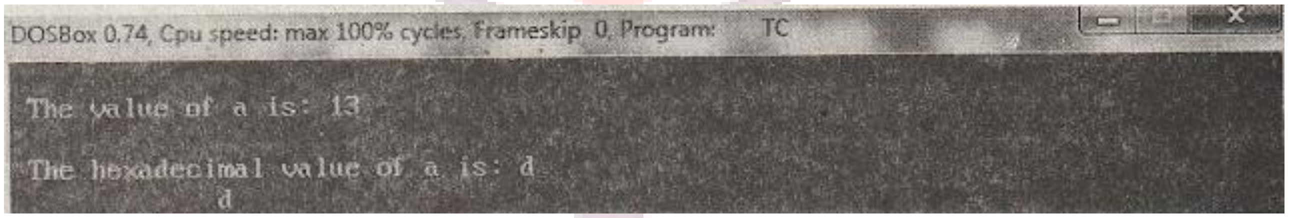
Generally, formatting is done using manipulators together with I/O functions.

Program

#include <iostream.h>

```
#include <conio.h>
int main()
{
clrscr();
int a = 13;
cout <<"\n The value of a is " <<a<<endl;  // displays decimal number 13
cout <<"\n The hexadecimal value of a is " <<hex<<a; // displays hexadecimal number of 13 i.e., d
cout <<endl;
cout.width(14); //displays the number with a width of 14
cout <<a;
getch();
return 0;
}
```

Output


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

```
The value of a is: 13

The hexadecimal value of a is: d
             d
```

The above program formats the data and displays onto the screen based on user requirement.

Formatted Input/Output

Formatted console I/O functions used in C++ for formatting the cae.

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()

(iy) 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. This is because, if in case the width size of a field is smaller than the size of the value that is to be printed, then C++ widens the field width to fit the value.

(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 éither 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()

It is a member 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'

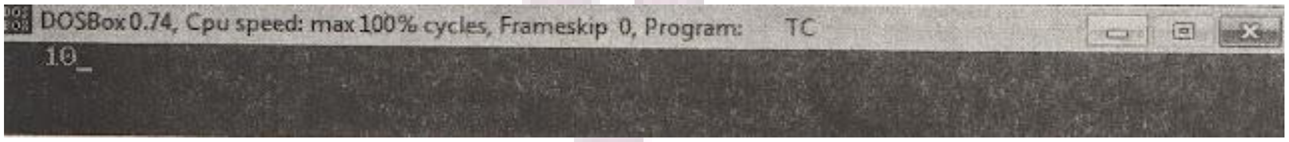
(b) DataType self (datatype)

This function on its invocation sets the flag in accordance to the bits marked in the parameterized 'data type.

Program

```
#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, Frameskip 0, Program:    TC
 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 bitfield 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::adjustfied 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 setf() 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).

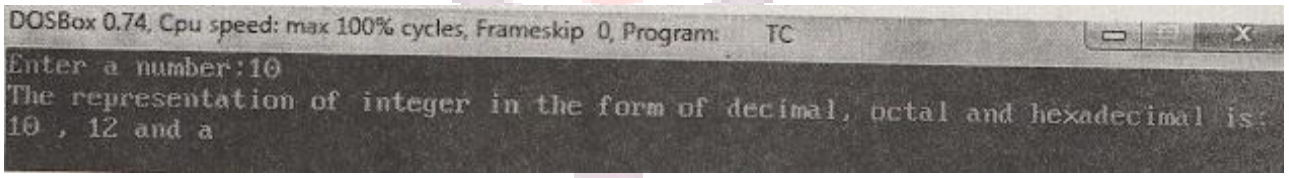
The declaration of ios::basefield can be done in the following manner,

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.setf(ios::dec, ios::basefield);
cout<<number<<" ,";
cout.setf(ios::oct, ios::basefield);
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
```

UNIT- V

Exception Handling: Benefits of exception handling, Throwing an exception, The try block, Catching an exception, Exception objects, Exception specifications, Stack unwinding, Rethrowing an exception, Catching all exceptions.

5.Exception Handling:

Exception :

An exception is a run time error that occurs while executing a program. The run time errors might be conditions such as division by zero, access to an array out of bounds, running out of memory or disk space etc. These exceptions should be handled correctly otherwise, the program terminates abnormally.

Exception Handling :

The process of handling these types of errors in C++ is known as exception handling.

In C++, we handle exceptions with the help of the try and catch blocks, along with the throw keyword.

- try - code that may raise an exception
- throw - throws an exception when an error is detected
- catch - code that handles the exception thrown by the throw keyword

Note: The throw statement is not compulsory, especially if we use standard C++ exceptions.

Syntax for Exception Handling in C++:

The basic syntax for exception handling in C++ is given below:

```
try {
   // code that may raise an exception
   throw argument;
}
catch (exception) {
   // code to handle exception
```

}

Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by the catch block.

When an exception occurs, the throw statement throws an exception, which is caught by the catch block.

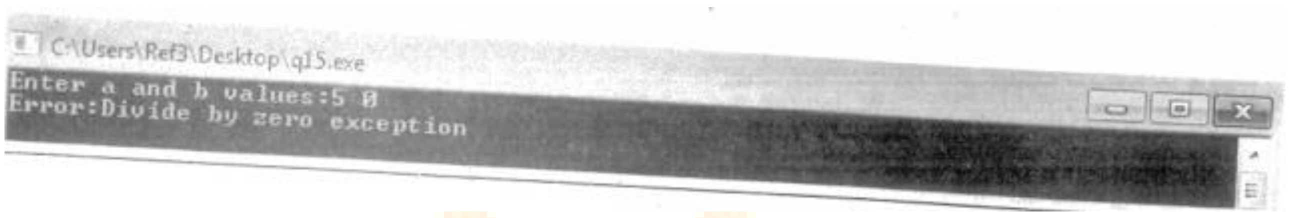
The catch block cannot be used without the try block.Example:

The following example illustrates the mechanism of exception handling.

```
#include<iostream>

using namespace std;

int main()

int a, b, c;

cout <<"Enter a and b values:";

cin >> a >>b;

if (b == 0)

try

{

if (b == 0) throw b;

else

cout <<"c="<<a/b;

}

catch(int x)

{

cout<<"Error: Divide by Zero exception\n";

}

return 0;
```

```
}
```

Output



```
Enter a and b values:5 0
Error:Divide by zero exception
```

Example 1: C++ Exception Handling

```
// program to divide two numbers

// throws an exception when the divisor is 0

#include <iostream>
using namespace std;

int main() {

   double numerator, denominator, divide;

   cout << "Enter numerator: ";
   cin >> numerator;

   cout << "Enter denominator: ";
   cin >> denominator;

   try {

      // throw an exception if denominator is 0
      if (denominator == 0)
         throw 0;

      // not executed if denominator is 0
      divide = numerator / denominator;
      cout << numerator << " / " << denominator << " = " << divide << endl;
   }

   catch (int num_exception) {
      cout << "Error: Cannot divide by " << num_exception << endl;
   }
```

```
    return 0;
}
```

Output 1:

```
Enter numerator: 72

Enter denominator: 0

Error: Cannot divide by 0
```

Output 2:

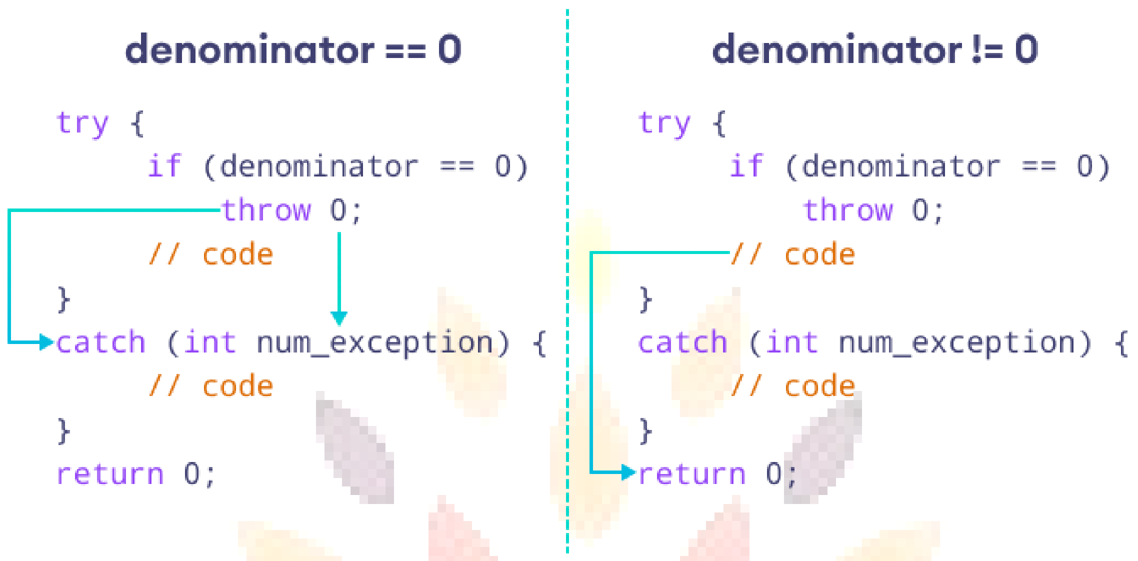
```
Enter numerator: 72

Enter denominator: 3

72 / 3 = 24
```

The above program divides two numbers and displays the result. But an exception occurs if the denominator is 0.

To handle the exception, we have put the code divide = numerator / denominator; inside the try block. Now, when an exception occurs, the rest of the code inside the try block is skipped.

The catch block catches the thrown exception and executes the statements inside it.

If none of the statements in the try block generates an exception, the catch block is skipped.



```
try {
    if (denominator == 0)
        throw 0;
    // code
}
catch (int num_exception) {
    // code
}
return 0;
```

Working of try, throw, and catch statements in C++:

Notice that we have thrown the int literal 0 with the code throw 0;.

We can throw any literal or variable or class, depending on the situation and depending on what we want to execute inside the catch block.

The catch parameter int num_exception takes the value passed by the throw statement i.e. the literal 0.

5.1Benefits of exception handling:

The benefits of exception handling are as follows

(a) Exception handling can control run tune errors that occur in the program.

(b) It can avoid abnormal termination of the program and also shows the behaviour of program to users.

(c) It can provide a facility to handle exceptions throws message regarding exception and completes the execution of program by catching the exception

(d) It can separate the error handling code and normal code by using try-catch block.

(e) It can produce the normal execution flow for a program.

(f) It can implement a clean way to propagate error.i.e.When an invoking method cannot manage a particular situations, then it throws an exception and asks the invoking method to deal with such situation.

(g) It develops a powerful coding which ensures that the exceptions can be prevented.

(h) It also allows to handle related exceptions by single exception handler. All the related errors are grouped together by using exceptions. And then they are handled by using single exception handler.

(i) Separation of annual code from error handling code eliminates the need for checking the errors in normal execution path there decreasing the cycles.

(j) A failed constructor can also be handled by throwing an exception. The code for try, catch and throw blocks must be written in the constructor itself.

5.2 Throwing an exception:

An exception detected in, try block is thrown using "throw" keyword. An exception can be thrown using in following number of ways.

throw(exception),

throw exception;

throw;

Where, exception is an object of any type including a constant. The third form of Throw statement is used in rethrowing of an exception. The objects that are intended for error handling can also be thrown.

The point at which an exception is thrown is called a throw point. When exception is thrown the control leaves the try block and it reaches to the catch block associated with the try block where the exception is handled.

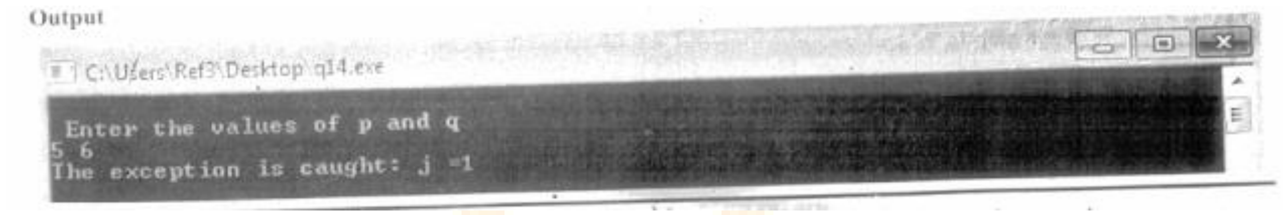
The throw point can he in a nested function call or in a nested scope within a try block. In any one of these cases the control is transferred to the catch statement

Program

```
#include<iostream>

using namespace

int main()
```

```
{
int p,q;
cout<<"\nEnter the values of p and q:\n";
cin>>p>>q;
int j;
j=p>q?0:1; //condition operator that determines whether p>q, if yes assign j=0 else j=1
try
{
if(j==0) //condition that verifies whether j-0 or not
{
cout<<"substration of (p-q)<<"\n"; //perform sbustration j=0
}
else
{
throw(j); //throw exception if j?0
}
}
catch(int i)
{
cout<<"the exception is caught: j="<<j<<"\n"; //display the detected exception on screen
}
return 0;
}
```

Output



```
Enter the values of p and q
5 6
The exception is caught: j =1
```

5.3 The try block:

Try is a keyword that is used to detect the exceptions i.e.., run time errors. The statements that generates exceptions are kept inside the try block. The runtime errors can be handled and prevented using try block. The general syntax of try block is as follows,

try

{

//code

throw exception

}

A try block can throw more than one exception. There should he a catch block to handle the exceptions thrown by try block.

Example

#include<iostream>

using namespace std;

void square( ) //A function called square is defined

{

int num;

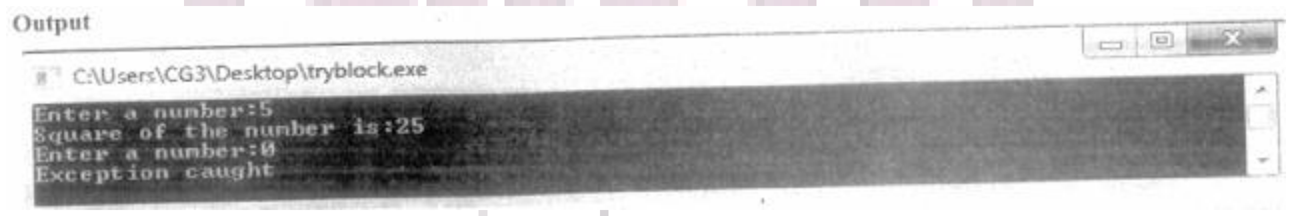
cout<<"Enter a number:";

cin>>num;

if(num>0) //This condition checks whether number is greater than zero or not

```
{
cout<<"Square of the number is:" <<num*num<endl; //Compute the square of number and
display the result on screen
else
throw(num); //if number<0 throw an exception.
int main()
{
try
{
square();
square();
}
catch(int i) //This statement catches an exception
{
cout<<"Exception caught\n":
}
return 0:
}
```

Output


Output

C:\Users\CG3\Desktop\tryblock.exe

```
Enter a number:5
Square of the number is:25
Enter a number:0
Exception caught
```

5.4 Catching an exception:

The catch block handles an exception thrown by the try The general syntax of catch block is shown below,

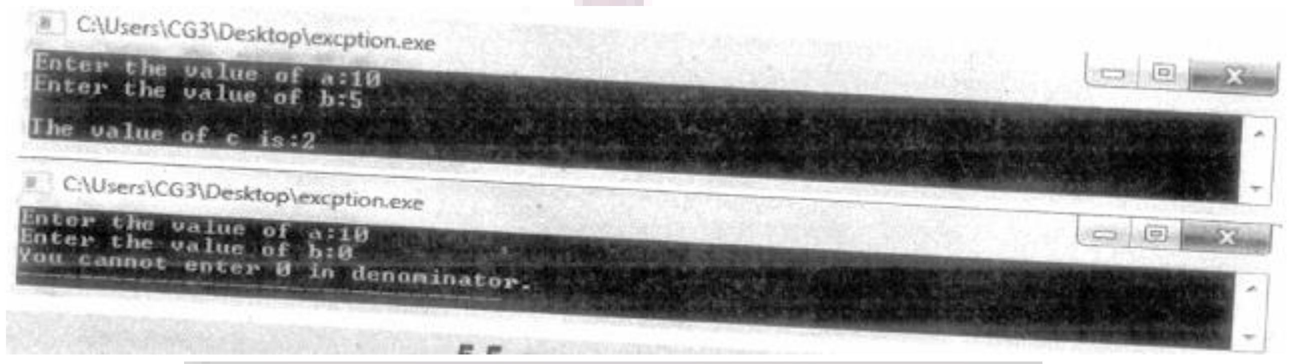
```
catch( type argument)

{

//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 immediately to the next statement following the catch block.

Program

```
#include<iostream>

#include<string>

using namespace std:

int main()

{

int a, b, c;

cout <<"Enter the value of a: ";

cin:>>a;

cout "Enter the value of b:;

cin>>b;

try

{

if(b)=0)

{

throw b;

}
else
if (b < 0)
{
throw "Negative denominator not allowed";
}
c = a/b;
cout<<"\nThe value of c is:" <<c;
}
catch(int num)
{
cout<<"You cannot enter "<<num<<" in denominator.";
}
}
```

Output



```
C:\Users\CG3\Desktop\excption.exe
Enter the value of a:10
Enter the value of b:5

The value of c is:2
```

```
C:\Users\CG3\Desktop\excption.exe
Enter the value of a:10
Enter the value of b:0
You cannot enter 0 in denominator.
```

5.5 Exception objects:

The exception object holds the error information about the exception that had occurred. The information includes the type of error i.e., logic error or run time error and state of the program when the error occurred. An exception object is created as soon as exception occurs and it is passed to the corresponding catch block as a parameter. This catch block contains the code to catch the occurred exception. It uses the methods of exception object for handling the exceptions.

<u>Syntax :</u>

```
try

{

throw exception object;

}

catch (Exception &exceptionobject)

{

}
```

To retrieve the message attached to the exception, exception object uses a method called what (). This method returns the actual messages.

<u>Syntax for what () Method:</u>

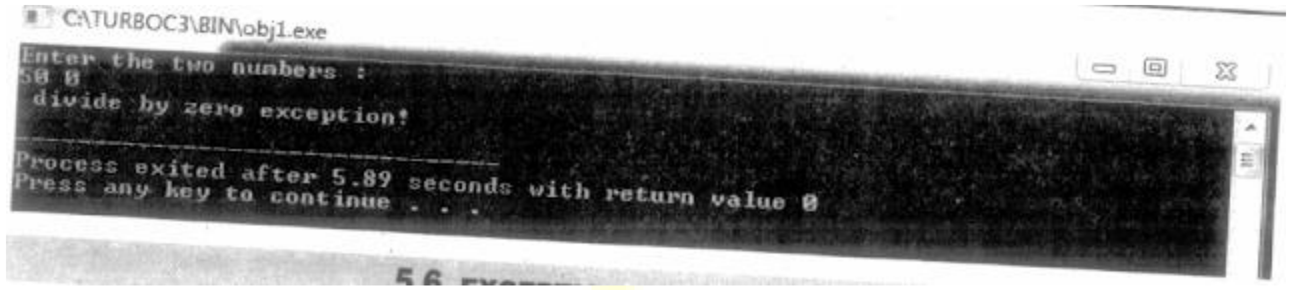
```
exceptionobject.what();
```

<u>Program :</u>

```
#include <iostream>

#include <exception>

using namespace std;

class MyException : public exception

{

public:

const char * what() const throw()

{

return" divide by zero exceptionl\n";

}

};
```

```
int main()
{
try
{
int x, y;
cout << "Enter the two numbers :. \n";
cin>>x>>y;
if(y==0)
{
MyException z;
throw z;
}
else
{
cout <<"x/y =" <<x/y << endl;
}
}
catch(exception &e)
{
cout<<e.what();
}
}
```

Output:


C:\TURBOC3\BIN\obj1.exe

```
Enter the two numbers :
50 0
 divide by zero exception!

--------------------------------
Process exited after 5.89 seconds with return value 0
Press any key to continue . . .
```

5.6 Exception specifications:

Exception specification is a feature which specifies the compiler about possible exception to be thrown by a function.

Syntax :

return_type FunctionName(arg_list) throw (type_list)

In the above syntax 'return_type' represents the type of the function. FunctionName represents the name of the function, arg_list indicates list of arguments passed to the function, throw is a keyword used to throw an exception by function and type_list indicates list of exception types. An empty exception specification can indicates that a function cannot throw any exception. if a function throws an exception which is not listed in exception specification then the unexpected function is called. This function terminates the program. In the function declaration, if the function does not have any exception specification then it can throw any exception.

Example :

1. void verify(int p) throw(int)

In the above example, void is the return type, and verify() is the name of the function, int p is the argument. It is followed by exception specification i.e., throw(int), the function verify() is throws an exception of type 'int'.

2. void verify(int p) throw()

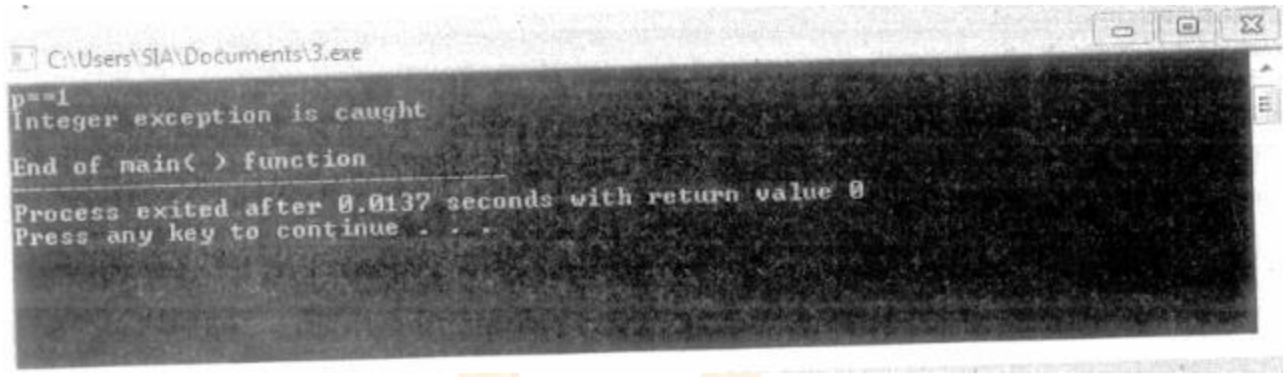
Here, the function can throw any exception because it has empty exception specification.

Program:

```
#include<iostream>

#include<exception>

using namespace std;

void verify(int p) throw(int)
```

```
{
if(p==1)
throw p;
cout<<"\nEnd of verify().function( )";
}
int main()
try
{
cout<<p==1\n";
verify(1);
}
catch(int i)
{
cout<<"Interger exception is caught \n";
}
cout<<"\nEnd of main() function";
//getch();
return 0;
}
```

Output


C:\Users\SIA\Documents\3.exe

```
p==1
Integer exception is caught

End of main( ) function
--------------------------------
Process exited after 0.0137 seconds with return value 0
Press any key to continue . . .
```

5.7 Stack unwinding:

Whenever exception is thrown, usually the control of program shifts from the try block and looks for a corresponding handler. During this session the C++ run time calls destructor relative to all automatic objects created while the initiation of try block. This consequence is usually referred as stack unwinding. As a result of it, all the automatic objects which were created get destroyed in the reverse order of creation.

Assume that the program control is busy in creating an object and this object in turn processes sub-objects or array elements. Now,Consider that an exception is raised while this process is in progress. To manage such consequences, destructors are called for those sub-objects array elements which were executed successfully prior to the raising of any exception.

The terminate function is called on the account that the destructor throws- an exception and there is none to handle this exception, while the stack unwinding process is in progress.

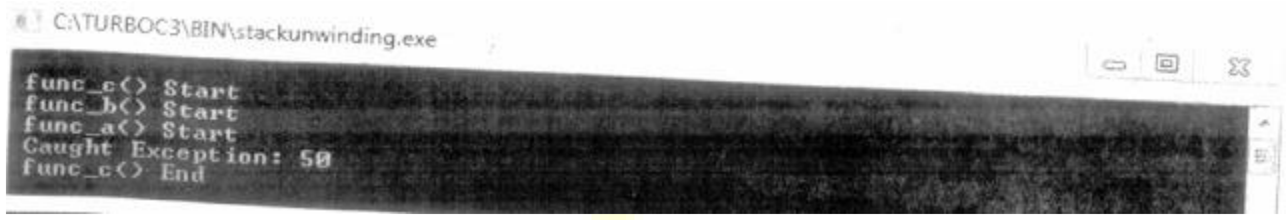
Program:

The following program demonstrates stack unwinding

```
#include<iostream>

using namespace std;

void func_a() throw (int)

{

cout<<"\n func_a() Start ";

throw 50;

cout<<"\nfunc_a() End ";

}
```

```
void func_b() throw(int)

{

cout<<"\n func_b() Start";

func_a();

cout<<"\n func_b() End";

}

void func_c() throw(int)

{

cout<<"\n func_c Start";

try

{

func_b();

}

catch(int i)

{

cout<<"\n Caught Exception:"<<i;

}

cout<<\n func_c() End";

}

int main()

{

func_c();

getchar();

return 0;

}
```

Output:



```
func_c() Start
func_b() Start
func_a() Start
Caught Exception: 50
func_c() End
```

5.8 Rethrowing an exception:

In the program execution, when an exception received by catch block is passed to another exception handler then such situation is referred to as rethrowing of exception.

This is done with the help of following statement,

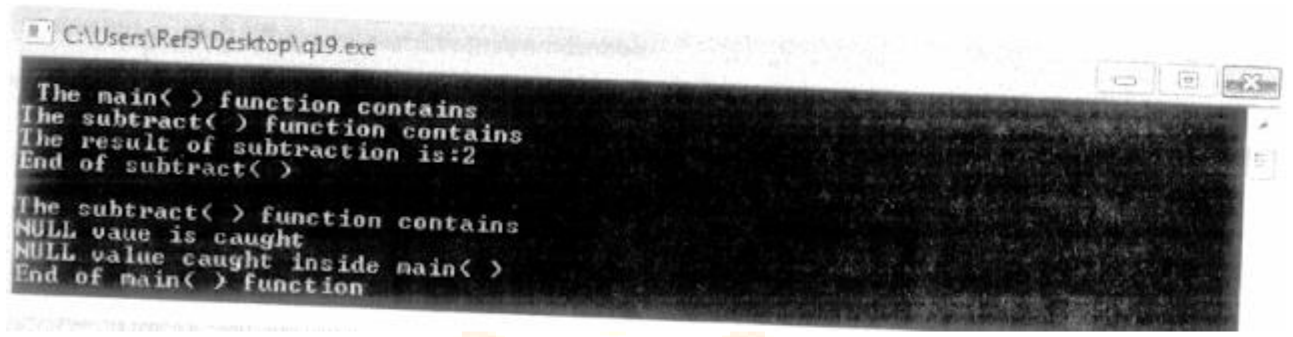
throw;

The above statement does not contain any arguments. This statement throws the exception to next try catch block.

Program:

```
#include<iostrearn>

using namespace std;

void subtract(int p, int q) //A function called subtract with two arguments is defined

{

cout<<"The subtract() function contains\n";

try //try block1

{

if(p==0) //This condition checks whether p is equal to zero or not.

//if yes throw an exception else perform subtraction

throw p; //This statement throws an exception

else

cout<<"The result of subtraction is:" <<p-q<<"\n";
}
catch(int) //This statement catches the exception throw it again to the next try block
{
cout<<"NULL value is caught\n";
throw;.
}
cout<<."End of subtract()\n\n";
}
int main( )
{
Cout<<"\n The main() function contains\n";
try
{
subtract(5, 3); //passing integer values to subtract
subtract(0, 2);
}
catch(int) //This statement catches the rethrown exception
{
cout<<"NULL value caught inside main()\n";
}
cout<<End of main() function \n";
return 0:
}
```

Output :


C:\Users\Ref3\Desktop\q19.exe

```
The main( ) function contains
The subtract( ) function contains
The result of subtraction is:2
End of subtract( )

The subtract( ) function contains
NULL vaue is caught
NULL value caught inside main( )
End of main( ) function
```

5.9 Catching all exceptions:

Catching Multiple Exceptions:

In C++, a user can catch all exceptions simultaneously In order 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:

catch

{

//statements fur handling various exceptions

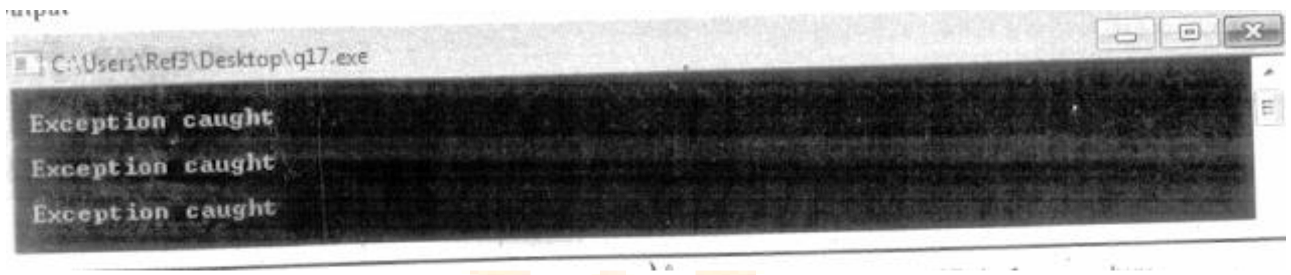
}

Program:

```
#include<iostream>
using namespace std;
void number(int p) //defining a function called number with single argument
{
try
{
 if(p==0)
{
//if p is equal to zero throw exception
throw 'p';
}
else
if(p>0)
{
throw 'q';
}
else
if(p<0)
{
throw 'r';
}
cout<<"Try Block \n";
}
catch(char ch) //defining single catch block
{
cout<<"\n Exception caught\n";
}
}
int main()
(
number(0);
number(5);
number(-2);
return 0;
```

}

Output



```
Exception caught
Exception caught
Exception caught
```