Syllabus

CS 35 OBJECT-ORIENTED PROGRAMMING 3 0 0 3

(Common to CSE & IT)

Aim: To understand the concepts of object-oriented programming and master OOP using C++.

UNIT I 9

Object oriented programming concepts – objects – classes – methods and messages – abstraction and encapsulation – inheritance – abstract classes – polymorphism.
Introduction to C++ – classes – access specifiers – function and data members – default arguments – function overloading – friend functions – const and volatile functions - static members – Objects – pointers and objects – constant objects – nested classes – local classes

UNIT II 9

Constructors – default constructor – Parameterized constructors – Constructor with dynamic allocation – copy constructor – destructors – operator overloading – overloading through friend functions – overloading the assignment operator – type conversion – explicit constructor

UNIT III 9

Function and class templates - Exception handling – try-catch-throw paradigm – exception specification – terminate and Unexpected functions – Uncaught exception.

UNIT IV 9

Inheritance – public, private, and protected derivations – multiple inheritance - virtual base class – abstract class – composite objects Runtime polymorphism – virtual functions – pure virtual functions – RTTI – typeid – dynamic casting – RTTI and templates – cross casting – down casting .

UNIT V 9

Streams and formatted I/O – I/O manipulators - file handling – random access – object serialization – namespaces - std namespace – ANSI String Objects – standard template library.

Total: 45

TEXT BOOKS:

1. B. Trivedi, “Programming with ANSI C++”, Oxford University Press, 2007.

REFERENCES:

1. Ira Pohl, “Object Oriented Programming using C++”, Pearson Education, Second Edition Reprint 2004..
2. S. B. Lippman, Josee Lajoie, Barbara E. Moo, “C++ Primer”, Fourth Edition, Pearson Education, 2005.
3. B. Stroustrup, “The C++ Programming language”, Third edition, Pearson Education, 2004.

Object Oriented Programming

Table of Contents

UNIT 1 5
Basic Concepts of OOPS 5
Objects 5
Classes 5
Inheritance 6
Data Abstraction 6
Data Encapsulation 6
Polymorphism 6
Overloading 6
Reusability 6
Variable, Constants and Data types in C++ 7
UNIT 2 22
Constructors and Destructors In C++ 22
Constructors: 22
What is the use of Constructor 22
General Syntax of Constructor 22
Default Constructor: 22
Parameterized Constructor: 23
Copy constructor; 23
Destructors 25
What is the use of Destructors? 25
General Syntax of Destructors 25
Operator Overloading 26
Unary Operators: 26
Binary Operators: 26
Operator Overloading – Unary operators 26
Operator Overloading – Binary Operators 29
Operator Overloading through friend functions 31
Overloading the Assignment Operator (=) 33
Type Conversions in C++ 35
What is Type Conversion 35
How to achieve this 35
Automatic Conversion otherwise called as Implicit Conversion 36
Type casting otherwise called as Explicit Conversion 36
Explicit Constructors 37
UNIT 3 39
Templates 39
Function templates 39
Class templates 42
Template specialization 44
Non-type parameters for templates 45

Exceptions 47
Exception specifications 49
UNIT 4 50
C++ Inheritance 50
What is Inheritance? 50
Features or Advantages of Inheritance 50
General Format for implementing the concept of Inheritance: 50
Types of Inheritance 53
Accessibility modes and Inheritance 53
Multiple Inheritance 53
Virtual Base Classes 55
Abstract Classes 56
Polymorphism 58
Static Polymorphism 58
Dynamic Polymorphism 59
Static Vs Dynamic Polymorphism 59
Introduction To Virtual Functions 59
Pointers to Derived Types 60
Virtual Functions 60
Virtual Functions and Inheritance 61
Rtti Constituents 62
std::type_info 62
dynamic_cast<> 64
Cross Casts 65
Downcasting from a Virtual Base 66
UNIT 5 69
C++ Standard Input Output Stream 69
What is a Stream? 69
Standard Input Stream 69
Ostream 70
Output Stream 70
Formatting information 71
State information 71
I/O Manipulators 72
File I/O with Streams 75
Object Serialization 79
Namespaces 79
Standard Template Library 84
ANSI String Class 85
Question Bank 92
PART A (2 Marks) 92
Unit I 92
Unit II 98
Unit III 101
Unit IV 104
Unit V 107

PART B - (16 Marks) 108
Unit I 108
Unit II 108
Unit III 109
Unit IV 109
Unit V 109

UNIT 1

Basic Concepts of OOPS

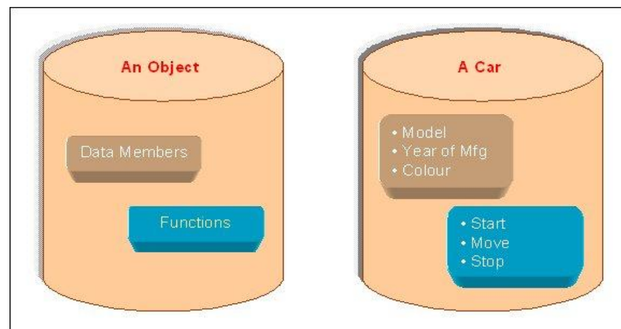
Before starting to learn C++ it is essential that one must have a basic knowledge of the concepts of Object Oriented Programming. Some of the important object oriented features are namely:

- Objects
- Classes
- Inheritance
- Data Abstraction
- Data Encapsulation
- Polymorphism
- Overloading
- Reusability

In order to understand the basic concepts in C++, the programmer must have a command of the basic terminology in object-oriented programming. Below is a brief outline of the concepts of Object-oriented programming languages:

Objects

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of an object. Each instance of an object can hold its own relevant data.



An Object is a collection of data members and associated member functions also known as methods.

Classes

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects. Characteristics of an object are represented in a class as **Properties (Attributes)**. The actions that can be performed by objects become functions of the class and are referred to as **Methods (Functions)**.

For example consider we have a Class of *Cars* under which *Santro Xing*, *Alto* and *WaganR* represents individual Objects. In this context each *Car* Object will have its own, Model, Year of Manufacture, Colour, Top Speed, Engine Power etc., which form **Properties** of the *Car* class and the associated actions i.e., object functions like Start, Move, Stop form the **Methods** of *Car* Class.

No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

Inheritance

Inheritance is the process of forming a new class from an existing class or *base class*. The base class is also known as *parent class* or *super class.* The new class that is formed is called *derived class*. Derived class is also known as a *child class* or *sub class.* Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

Data Abstraction

Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without presenting the details.

Data Encapsulation

Data Encapsulation combines data and functions into a single unit called Class. When using Data Encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class. Data Encapsulation enables the important concept of data hiding possible.

Polymorphism

Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways.

Overloading

Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an exiting operator or function begins to operate on new data type, or class, it is understood to be overloaded.

Reusability

This term refers to the ability for multiple programmers to use the same written and debugged existing class of data. This is a time saving device and adds code efficiency to

the language. Additionally, the programmer can incorporate new features to the existing class, further developing the application and allowing users to achieve increased performance.

Introduction to C++

Variable, Constants and Data types in C++

Variables

A variable is the storage location in memory that is stored by its value. A variable is identified or denoted by a variable name. The variable name is a sequence of one or more letters, digits or underscore, for example: character _

Rules for defining variable name:

- A variable name can have one or more letters or digits or underscore for example character _.
- White space, punctuation symbols or other characters are not permitted to denote variable name.
- A variable name must begin with a letter.
  .
- Variable names cannot be keywords or any reserved words of the C++ programming language.

C++ is a case-sensitive language. Variable names written in capital letters differ from variable names with the same name but written in small letters. For example, the variable name EXFORSYS differs from the variable name exforsys.

Data Types

Below is a list of the most commonly used *Data Types* in C++ programming language

short int	short integer.
int	integer.
long int	long integer.
float	floating point
double	double precision floating point number.
long double	double precision floating point number.
char	single character.
bool	boolean value. It can take one of two values True or False

Using variable names and data type, we shall now learn how to declare variables.

Declaring Variables:

In order for a variable to be used in C++ programming language, the variable must first be declared. The syntax for declaring variable names is

data type variable name;

The date type can be int or float or any of the data types listed above. A variable name is given based on the rules for defining variable name (refer above rules).

Example:
int a;

This declares a variable name a of type int.

If there exists more than one variable of the same type, such variables can be represented by separating variable names using comma.

For instance

int x,y,z ;

This declares 3 variables x, y and z all of data type int.

The data type using integers (int, short int, long int) are further assigned a value of **signed** or **unsigned**. Signed integers signify positive and negative number value. Unsigned integers signify only positive numbers or zero.

For example it is declared as

unsigned short int a;
signed int z;

By default, unspecified integers signify a signed integer.

For example:

int a;
is declared a signed integer
It is possible to initialize values to variables:

data type variable name = value;

Example:
int a=0;
int b=5;

Constants

Constants have fixed value. Constants, like variables, contain data type. Integer constants are represented as decimal notation, octal notation, and hexadecimal notation. Decimal notation is represented with a number. Octal notation is represented with the number preceded by a zero character. A hexadecimal number is preceded with the characters 0x.

Example

80 represent decimal
0115 represent octal
0x167 represent hexadecimal

By default, the integer constant is represented with a number.

The unsigned integer constant is represented with an appended character **u**. The long integer constant is represented with character **l**.

Example:

78 represent int
85u present unsigned int
78l represent long

Floating point constants are numbers with decimal point and/or exponent.
Example

2.1567
4.02e24

These examples are valid floating point constants.

Floating point constants can be represented with f for floating and l for double precision floating point numbers.

Character constants have single character presented between single quotes.

Example
'c'
'a'

are all character constants.

Strings are sequences of characters signifying string constants. These sequence of characters are represented between double quotes.

Example:

"Exforsys Training"
is an example of string constant.
Referencing variables
The & operator is used to reference an object. When using this operator on an object, you are provided with a pointer to that object. This new pointer can be used as a parameter or be assigned to a variable.

C++ Objects and Classes

An Overview about Objects and Classes

In object-oriented programming language C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an *object*. A *class* is an extended concept similar to that of *structure* in C programming language, this

class describes the data properties alone. In C++ programming language, *class* describes both the properties (data) and behaviors (functions) of objects. *Classes* are not *objects*, but they are used to instantiate *objects*.

Features of Class:

Classes contain member data and member functions. As a unit, the collection of member data and member functions is an object. Therefore, this unit of objects makes up a class.

How to write a Class:

In Structure in C programming language, a structure is specified with a name. The C++ programming language extends this concept. A class is specified with a name after the keyword class.

The starting flower brace symbol, {is placed at the beginning of the code. Following the flower brace symbol, the body of the class is defined with the member functions data. Then the class is closed with a flower brace symbol} and concluded with a colon;.

```
class exforsys
{
  member data;
  member functions;
  ……………
};
```

There are different access specifiers for defining the data and functions present inside a class.

Access specifiers:

Access specifiers are used to identify access rights for the data and member functions of the class. There are three main types of access specifiers in C++ programming language:

private
public
protected

- A *private* member within a class denotes that only members of the same class have accessibility. The *private* member is inaccessible from outside the class.
- *Public* members are accessible from outside the class.
- A protected access specifier is a stage between *private* and *public* access. If member functions defined in a class are *protected*, they cannot be accessed from outside the class but can be accessed from the derived class.

When defining access specifiers, the programmer must use the keywords: *private*, *public* or *protected* when needed, followed by a semicolon and then define the data and member functions under it.

```
class exforsys
{
```

```
        private:
        int x,y;
        public:
        void sum()
        {
          ………
          ………
        }
      };
```

In the code above, the member *x* and *y* are defined as private access specifiers. The member function sum is defined as a public access specifier.

General Syntax of a class:

General structure for defining a class is:

```
class classname
{
  acess specifier:
  data member;
  member functions;
acess specifier:
  data member;
  member functions;
};
```

Generally, in class, all members (data) would be declared as private and the member functions would be declared as public. Private is the default access level for specifiers. If no access specifiers are identified for members of a class, the members are defaulted to private access.

```
class exforsys
{
  int x,y;
  public:
  void sum()
  {
    ………
    ………
  }
};
```

In this example, for members x and y of the class *exforsys* there are no access specifiers identified. *exforsys* would have the default access specifier as private.

Creation of Objects:

Once the class is created, one or more objects can be created from the class as objects are instance of the class.

Juts as we declare a variable of data type *int* as:

int x;

Objects are also declared as:

class name followed by object name;
exforsys e1;
This declares *e1* to be an object of class exforsys.
For example a complete class and object declaration is given below:

```
class exforsys
{
  private:
  int x,y;
  public:
  void sum()
  {
    ………
    ………
  }
};
main()
{
  exforsys e1;
  ……………
  ……………
}
```

The object can also be declared immediately after the class definition. In other words the object name can also be placed immediately before the closing flower brace symbol } of the class declaration.

For example

```
class exforsys
{
  private:
  int x,y;
  public:
  void sum()
  {
    ………
    ………
  }
}e1 ;
```

The above code also declares an object ***e1*** of class exforsys.
It is important to understand that in object-oriented programming language, when a class is created no memory is allocated. It is only when an object is created is memory then allocated.

Function Overloading

A function is overloaded when same name is given to different function. However, the two functions with the same name will differ at least in one of the following.

a) The number of parameters
b) The data type of parameters
c) The order of appearance

These three together are referred to as the **function signature**.

For example if we have two functions :

void foo(int i,char a);
void boo(int j,char b);

Their signature is the same **(int ,char)** but a function

void moo(int i,int j) ; has a signature **(int, int)** which is different.

While **overloading a function**, the return type of the functions needs to be the same.

In general functions are overloaded when :
1. Functions differ in function signature.
2. Return type of the functions is the same.

Here s a basic example of **function overloading**

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

class arith {
public:
   void calc(int num1)

{
cout<<"Square of a given number: " <<num1*num1 <<endl;
}

   void calc(int num1, int num2 )
{
cout<<"Product of two whole numbers: " <<num1*num2 <<endl;
}
};

int main() //begin of main function
```

```
{
   arith a;
   a.calc(5);
   a.calc(6,7);
}
```

Let us see what we did in the **function overloading example**.

First the overloaded function in this example is **calc**. If you have noticed we have in our **arith** class two functions with the name **calc**. The fist one takes one integer number as a parameter and prints the square of the number. The second calc function takes two integer numbers as parameters, multiplies the numbers and prints the product. This is all we need for making a successful overloading of a function.

a) we have two functions with the same name : *calc*
b) we have different signatures : *(int) , (int, int)*
c) return type is the same : void

The result of the execution looks like this

```
Square of a given number: 25
Product of two whole numbers: 42
```

The result demonstrates the overloading concept. Based on the arguments we use when we call the **calc** function in our code :

```
a.calc(5);
a.calc(6,7);
```

The compiler decides witch function to use at the moment we call the function.

C++ Friend Functions

Need for Friend Function

As discussed in the earlier sections on access specifiers, when a data is declared as private inside a class, then it is not accessible from outside the class. A function that is not a member or an external class will not be able to access the private data. A programmer may have a situation where he or she would need to access private data from non-memberfunctions and external classes. For handling such cases, the concept of Friend functions is a useful tool.

What is a Friend Function?

A friend function is used for accessing the non-public members of a class. A class can allow non-member functions and other classes to access its own private data, by making them friends. Thus, a friend function is an ordinary function or a member of another class.

How to define and use Friend Function in C++?

The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend. The friend function must have the class to which it is declared as friend passed to it in argument.

Some important points to note while using friend functions in C++:

- The keyword friend is placed only in the function declaration of the friend function and not in the function definition.
- It is possible to declare a function as friend in any number of classes.
- When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend.
- A friend function, even though it is not a member function, would have the rights to access the private members of the class.
- It is possible to declare the friend function as either private or public.
- The function can be invoked without the use of an object. The friend function has its argument as objects, seen in example below.

Example to understand the friend function:

```
#include <iostream.h>
class exforsys
{
private:
int a,b;
public:
void test()
{
a=100;
b=200;
}
friend int compute(exforsys e1)

//Friend Function Declaration with keyword friend and with the object of class exforsys
to which it is friend passed to it
};


int compute(exforsys e1)
{
//Friend Function Definition which has access to private data
return int(e1.a+e2.b)-5;
}

main()
{
exforsys e;
```

```
e.test();
cout<<"The result is:"<
//Calling of Friend Function with object as argument.
}
```

The output of the above program is
The result is:295

The function compute() is a non-member function of the class exforsys. In order to make this function have access to the private data a and b of class exforsys , it is created as afriend function for the class exforsys. As a first step, the function compute() is declared as friend in the class exforsys as:

friend int compute (exforsys e1)

The keyword friend is placed before the function. The function definition is written as a normal function and thus, the function has access to the private data a and b of the class exforsys. It is declared as friend inside the class, the private data values a and b are added, 5 is subtracted from the result, giving 295 as the result. This is returned by the function and thus the output is displayed as shown above.

Constant and volatile member functions

A member function declared with the **const** qualifier can be called for constant and nonconstant objects. A nonconstant member function can only be called for a nonconstant object. Similarly, a member function declared with the **volatile** qualifier can be called for volatile and nonvolatile objects. A nonvolatile member function can only be called for a nonvolatile object.

static members

Class members can be declared using the storage class specifier static in the class member list. Only one copy of the static member is shared by all objects of a class in a program. When you declare an object of a class having a static member, the static member is not part of the class object.

A typical use of static members is for recording data common to all objects of a class. For example, you can use a static data member as a counter to store the number of objects of a particular class type that are created. Each time a new object is created, this static data member can be incremented to keep track of the total number of objects.

You access a static member by qualifying the class name using the :: (scope resolution) operator. In the following example, you can refer to the static member f() of class type X as X::f() even if no object of type X is ever declared:

```
class X {
static int f();
};

int main() {
X::f();
```

}

Pointers to classes

It is perfectly valid to create pointers that point to classes. We simply have to consider that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer. For example:

CRectangle * prect;

is a pointer to an object of class CRectangle.

As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (->) of indirection. Here is an example with some possible combinations:

```
// pointer to classes example
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area (void) {return (width * height);}
};

void CRectangle::set_values (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle a, *b, *c;
  CRectangle * d = new CRectangle[2];
  b= new CRectangle;
  c= &a;
  a.set_values (1,2);
  b->set_values (3,4);
  d->set_values (5,6);
  d[1].set_values (7,8);
  cout << "a area: " << a.area() << endl;
  cout << "*b area: " << b->area() << endl;
  cout << "*c area: " << c->area() << endl;
  cout << "d[0] area: " << d[0].area() << endl;
  cout << "d[1] area: " << d[1].area() << endl;
  delete[] d;
  delete b;
```

```
  return 0;
}
```

Output:

```
a area: 2
*b area: 12
*c area: 2
d[0] area: 30
d[1] area: 56
```

Next you have a summary on how can you read some pointer and class operators (*, &, ., ->, [ ]) that appear in the previous example:

Expression	Can be read as
*x	pointed by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed by x
(*x).y	member y of object pointed by x (equivalent to the previous one)
X[0]	first object pointed by x
X[1]	second object pointed by x
X[n]	(n+1)th object pointed by x

Difference between const variables and const object

Constant variables are the variables whose value cannot be changed through out the programme but if any object is constant, value of any of the data members (const or non const) of that object cannot be changed through out the programme. Constant object can invoke only constant function.

Nested classes

A nested class is declared within the scope of another class. The name of a nested class is local to its enclosing class. Unless you use explicit pointers, references, or object names, declarations in a nested class can only use visible constructs, including type names, static members, and enumerators from the enclosing class and global variables.

Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes. Member functions of the enclosing class have no special access to members of a nested class. The following example demonstrates this:

```
class A {
int x;
```

```
class B { };
class C {
      // The compiler cannot allow the following
      // declaration because A::B is private:
    //   B b;

    int y;
    void f(A* p, int i) {

    // The compiler cannot allow the following
    // statement because A::x is private:
    //   p->x = i;

    }
  };

void g(C* p) {
      // The compiler cannot allow the following
      // statement because C::y is private:
      //   int z = p->y;
  }
};

int main() { }
```

The compiler would not allow the declaration of object b because class A::B is private. The compiler would not allow the statement p->x = i because A::x is private. The compiler would not allow the statement int z = p->y because C::y is private.

Local classes

A local class is declared within a function definition. Declarations in a local class can only use type names, enumerations, static variables from the enclosing scope, as well as external variables and functions.

For example:

```
int x;                         // global variable
void f()                         // function definition
{
      static int y;            // static variable y can be used by
                           // local class
      int x;                   // auto variable x cannot be used by
                           // local class
      extern int g();          // extern function g can be used by
         // local class
```

```
  class local            // local class
    {
      int g() { return x; }      // error, local variable x
// cannot be used by g
        int h() { return y; }      // valid,static variable y
        int k() { return ::x; }    // valid, global x
        int l() { return g(); }    // valid, extern function g
    };
}

int main()
{
  local* z;              // error: the class local is not visible
    // ...}
```

Member functions of a local class have to be defined within their class definition, if they are defined at all. As a result, member functions of a local class are inline functions. Like all member functions, those defined within the scope of a local class do not need the keyword inline.

A local class cannot have static data members. In the following example, an attempt to define a static member of a local class causes an error:

```
void f()
{
  class local
   {
     int f();            // error, local class has noninline
                        // member function
     int g() {return 0;}   // valid, inline member function
     static int a;       // error, static is not allowed for
                        // local class
     int b;              // valid, nonstatic variable
   };
}
//     . . .
```

An enclosing function has no special access to members of the local class.

UNIT 2

Constructors and Destructors In C++

Constructors:

What is the use of Constructor

The main use of constructors is to initialize objects. The function of initialization is automatically carried out by the use of a special member function called a constructor.

General Syntax of Constructor

Constructor is a special member function that takes the same name as the class name. The syntax generally is as given below:

<class name> { arguments};

The default constructor for a class X has the form

X::X()
In the above example the arguments is optional.
The constructor is automatically invoked when an object is created.

The various types of constructors are

- Default constructors
- Parameterized constructors
- Copy constructors

Default Constructor:

This constructor has no arguments in it. Default Constructor is also called as *no argument constructor*.

For example:

```
Class Exforsys
{
   private:
      int a,b;
   public:
      Exforsys(); //default
Constructor
      ...
};

Exforsys :: Exforsys()
```

```
{
   a=0;
   b=0;
}
```

Parameterized Constructor:

A parameterized constructor is just one that has parameters specified in it.

Example:

```
class Exforsys
{
   private:
      int a,b;
   public:
      Exforsys(int,int);// Parameterized constructor
      ...
};

Exforsys :: Exforsys(int x, int y)
{
   a=x;
   b=y;
}
```

Copy constructor;

One of the more important forms of an overloaded constructor is the copy constructor. The purpose of the copy constructor is to initialize a new object with data copied from another object of the same class.

For example to invoke a copy constructor the programmer writes:

```
Exforsys e3(e2);
or
Exforsys e3=e2;
```

Both the above formats can be used to invoke a copy constructor.

For Example:

```
#include <iostream.h>
class Exforsys()
{
   private:
      int a;
   public:
      Exforsys()
      { }
      Exforsys(int w)
   {
      a=w;
   }
   Exforsys(Exforsys& e)
   {
      a=e.a;
      cout<<" Example of Copy
Constructor";
   }
   void result()
   {
      cout<< a;
   }
};

void main()
{
   Exforsys e1(50);
   Exforsys e3(e1);
   cout<< "\ne3=";e3.result();
}
```

In the above the copy constructor takes one argument an object of type Exforsys which is passed by reference. The output of the above program is

Example of Copy Constructor
e3=50

Some important points about constructors:

- A constructor takes the same name as the class name.
- The programmer cannot declare a constructor as virtual or static, nor can the programmer declare a constructor as const, volatile, or const volatile.

- No return type is specified for a constructor.
- The constructor must be defined in the public. The constructor must be a public member.
- Overloading of constructors is possible.

Destructors

What is the use of Destructors?

Destructors are also special member functions used in C++ programming language. Destructors have the opposite function of a constructor. The main use of destructors is to release dynamic allocated memory. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically called when an object is destroyed. Like constructors, destructors also take the same name as that of the class name.

General Syntax of Destructors

~ classname();

The above is the general syntax of a destructor. In the above, the symbol tilda ~ represents a destructor which precedes the name of the class.

Some important points about destructors:

- Destructors take the same name as the class name.
- Like the constructor, the destructor must also be defined in the public. The destructor must be a public member.
- The Destructor does not take any argument which means that destructors cannot be overloaded.
- No return type is specified for destructors.

For example:

```
class Exforsys
{
   private:
      ……………
   public:
      Exforsys()
      { }
      ~ Exforsys()
      { }
}
```

Operator Overloading

Operator overloading is a very important feature of Object Oriented Programming. It is because by using this facility programmer would be able to create new definitions to existing operators. In other words a single operator can perform several functions as desired by programmers.

Operators can be broadly classified into:

- Unary Operators
- Binary Operators

Unary Operators:

As the name implies takes operate on only one operand. Some unary operators are namely

++ - Increment operator

-- - Decrement Operator

! - Not operator

• unary minus.

Binary Operators:

The arithmetic operators, comparison operators, and arithmetic assignment operators come under this category.

Both the above classification of operators can be overloaded. So let us see in detail each of this.

Operator Overloading – Unary operators

As said before operator overloading helps the programmer to define a new functionality for the existing operator. This is done by using the keyword **operator**.

The general syntax for defining an operator overloading is as follows:

```
return_type classname :: operator operator symbol(argument)
{
…………..
statements;
}
```

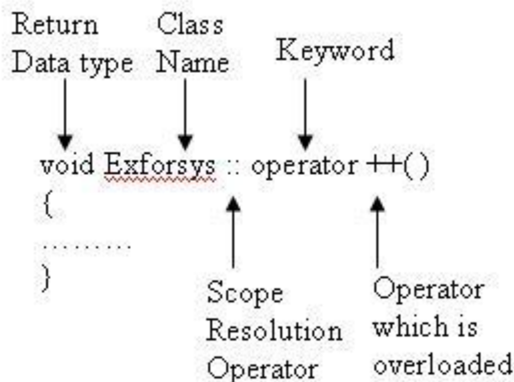
Thus the above clearly specifies that operator overloading is defined as a member function by making use of the keyword operator.

In the above:

- return_type – is the data type returned by the function
- class name - is the name of the class
- operator – is the keyword
- operator symbol – is the symbol of the operator which is being overloaded or defined for new functionality
- **::** - is the scope resolution operator which is used to use the function definition outside the class.

For example

Suppose we have a class say Exforsys and if the programmer wants to define a operator overloading for unary operator say ++, the function is defined as



Inside the class Exforsys the data type that is returned by the overloaded operator is defined as

```
class Exforsys
{
   private:
   ……….
   public:
   void operator ++( );
   …………
};
```

The important steps involved in defining an operator overloading in case of unary operators are namely:

- Inside the class the operator overloaded member function is defined with the return data type as member function or a friend function.
- If the function is a member function then the number of arguments taken by the operator member function is none.
- If the function defined for the operator overloading is a friend function then it takes one argument.

Now let us see how to use this overloaded operator member function in the program

```
#include <iostream.h>
class Exforsys
{
   private:
   int x;
   public:
   Exforsys( ) { x=0; }       //Constructor
   void display();
   void Exforsys ++( );           //overload unary ++
};

void Exforsys :: display()
{
   cout<<"\nValue of x is: " << x;
}

void Exforsys :: operator ++( ) //Operator Overloading for operator ++
                                              defined
{
   ++x;
}

void main( )
{
   Exforsys e1,e2;         //Object e1 and e2 created
   cout<<"Before Increment"
   cout <<"\nObject e1: "<<e1.display();
   cout <<"\nObject e2: "<<e2.display();
   ++e1;  //Operator overloading applied
   ++e2;
   cout<<"\n After Increment"
   cout <<"\nObject e1: "<<e1.display();
   cout <<"\nObject e2: "<<e2.display();
}
```

The output of the above program is:

```
Before Increment
Object e1:
Value of x is: 0
Object e1:
Value of x is: 0
Before Increment
Object e1:
Value of x is: 1
Object e1:
Value of x is: 1
```

In the above example we have created 2 objects e1 and e2 f class Exforsys. The operator ++ is overloaded and the function is defined outside the class Exforsys.

When the program starts the constructor Exforsys of the class Exforsys initialize the values as zero and so when the values are displayed for the objects e1 and e2 it is displayed as zero. When the object ++e1 and ++e2 is called the operator overloading function gets applied and thus value of x gets incremented for each object separately. So now when the values are displayed for objects e1 and e2 it is incremented once each and gets printed as one for each object e1 and e2.

Operator Overloading – Binary Operators

Binary operators, when overloaded, are given new functionality. The function defined for binary operator overloading, as with unary operator overloading, can be member function or friend function.

The difference is in the number of arguments used by the function. In the case of binary operator overloading, when the function is a member function then the number of arguments used by the operator member function is one (see below example). When the function defined for the binary operator overloading is a friend function, then it uses two arguments.

Binary operator overloading, as in unary operator overloading, is performed using a keyword operator.

Binary operator overloading example:

```
#include <iostream.h>
class Exforsys
{
private:
int x;
int y;
```

```
public:
Exforsys()                  //Constructor
{ x=0; y=0; }

void getvalue( )            //Member Function for Inputting Values
{
cout << "\n Enter value for x: ";
cin >> x;
cout << "\n Enter value for y: ";
cin>> y;
}

void displayvalue( )        //Member Function for Outputting Values
{
cout <<"value of x is: " << x <<"; value of y is: "<<y
}



Exforsys operator +(Exforsys);
};

Exforsys Exforsys :: operator + (Exforsys e2)
//Binary operator overloading for + operator defined
{
int x1 = x+ e2.x;
int y1 = y+ e2.y;
return Exforsys(x1,y1);
}

void main( )
{
Exforsys e1,e2,e3;          //Objects e1, e2, e3 created
cout<<\n"Enter value for Object e1:";
e1.getvalue( );
cout<<\n"Enter value for Object e2:";
e2.getvalue( );
e3= e1+ e2;                 //Binary Overloaded operator used
cout<< "\nValue of e1 is:"<<e1.displayvalue();
cout<< "\nValue of e2 is:"<<e2.displayvalue();
cout<< "\nValue of e3 is:"<<e3.displayvalue();
}
```

The output of the above program is:

Enter value for Object e1:
Enter value for x: 10
Enter value for y: 20
Enter value for Object e2:
Enter value for x: 30
Enter value for y: 40
Value of e1 is: value of x is: 10; value of y is: 20
Value of e2 is: value of x is: 30; value of y is: 40
Value of e3 is: value of x is: 40; value of y is: 60

In the above example, the class Exforsys has created three objects e1, e2, e3. The values are entered for objects e1 and e2. The binary operator overloading for the operator '+' is declared as a member function inside the class Exforsys. The definition is performed outside the class Exforsys by using the scope resolution operator and the keyword operator.

The important aspect is the statement:

```
e3= e1 + e2;
```

The binary overloaded operator '+' is used. In this statement, the argument on the left side of the operator '+', e1, is the object of the class Exforsys in which the binary overloaded operator '+' is a member function. The right side of the operator '+' is e2. This is passed as an argument to the operator '+' . Since the object e2 is passed as argument to the operator'+' inside the function defined for binary operator overloading, the values are accessed as e2.x and e2.y. This is added with e1.x and e1.y, which are accessed directly as x and y. The return value is of type class Exforsys as defined by the above example.

There are important things to consider in operator overloading with C++ programming language. Operator overloading adds new functionality to its existing operators. The programmer must add proper comments concerning the new functionality of the overloaded operator. The program will be efficient and readable only if operator overloading is used only when necessary.

Some operators cannot be overloaded:

- Scope resolution operator denoted by ::
- Member access operator or the dot operator denoted by .
- Conditional operator denoted by ?:
- Pointer to member operator denoted by .*

Operator Overloading through friend functions

```
// Using friend functions to
// overload addition and subtarction
// operators
#include <iostream.h>

class myclass
  {
int a;
int b;

public:
myclass(){}
myclass(int x,int y){a=x;b=y;}
void show()
    {
cout<<a<<endl<<b<<endl;
    }

// these are friend operator functions
// NOTE: Both the operans will be be
// passed explicitely.
// operand to the left of the operator
// will be passed as the first argument
// and operand to the right as the second
// argument
friend myclass operator+(myclass,myclass);
friend myclass operator-(myclass,myclass);

  };

myclass operator+(myclass ob1,myclass ob2)
  {
myclass temp;
temp.a = ob1.a + ob2.a;
temp.b = ob1.b + ob2.b;

return temp;
  }

myclass operator-(myclass ob1,myclass ob2)
  {
myclass temp;
temp.a = ob1.a - ob2.a;
temp.b = ob1.b - ob2.b;

return temp;
```

```
  }

void main()
  {
myclass a(10,20);
myclass b(100,200);

a=a+b;
a.show();
  }
```

Overloading the Assignment Operator (=)

We know that if we want objects of a class to be operated by common operators then we need to *overload them*. But there is one operator whose operation is automatically crested by C++ for every class we define, it is the assignment operator '='.

Actually we have been using similar statements like the one below previously

ob1=ob2;

where ob1 and ob2 are objects of a class.

This is because even if we don't overload the '=' operator, the above statement is valid.

because C++ automatically creates a default assignment operator. The default operator created, does a member-by-member copy, but if we want to do something specific we may overload it.

The simple program below illustrates how it can be done. Here we are defining two similar classes, one with the default assignment operator (created automatically) and the other with the overloaded one. Notice how we could control the way assignments are done in that case.

```
// Program to illustrate the
// overloading of assignment
// operator '='
#include <iostream.h>

// class not overloading the
// assignment operator
class myclass
  {
int a;
int b;
```

```
public:
myclass(int, int);
void show();
  };

myclass::myclass(int x,int y)
  {
a=x;
b=y;
  }

void myclass::show()
  {
cout<<a<<endl<<b<<endl;
  }

// class having overloaded
// assignment operator
class myclass2
  {
int a;
int b;

public:
myclass2(int, int);
void show();

myclass2 operator=(myclass2);
  };

myclass2 myclass2::operator=(myclass2 ob)
  {
// -- do something specific—
// this is just to illustrate
// that when overloading '='
// we can define our own way
// of assignment
b=ob.b;

return *this;
  };

myclass2::myclass2(int x,int y)
  {
a=x;
```

```
b=y;
  }

void myclass2::show()
  {
cout<<a<<endl<<b<<endl;
  }

// main
void main()
  {
myclass ob(10,11);
myclass ob2(20,21);

myclass2 ob3(100,110);
myclass2 ob4(200,210);

// does a member-by-member copy
// '=' operator is not overloaded
ob=ob2;
ob.show();


// does specific assignment as
// defined in the overloaded
// operator definition
ob3=ob4;
ob3.show();
  }
```

Type Conversions in C++

What is Type Conversion

It is the process of converting one type into another. In other words converting an expression of a given type into another is called type casting.

How to achieve this

There are two ways of achieving the type conversion namely:

Automatic Conversion otherwise called as **Implicit Conversion**
Type casting otherwise called as **Explicit Conversion**

Let us see each of these in detail:

Automatic Conversion otherwise called as Implicit Conversion

This is not done by any conversions or operators. In other words value gets automatically converted to the specific type in which it is assigned.

Let us see this with an example:

```
#include <iostream.h>
void main()
{
short x=6000;
int y;
y=x;
}
```

In the above example the data type short namely variable x is converted to int and is assigned to the integer variable y.

So as above it is possible to convert short to int, int to float and so on.

Type casting otherwise called as Explicit Conversion

Explicit conversion can be done using type cast operator and the general syntax for doing this is

datatype (expression);

Here in the above datatype is the type which the programmer wants the expression to gets changed as

In C++ the type casting can be done in either of the two ways mentioned below namely:

- C-style casting
- C++-style casting

The C-style casting takes the syntax as

(type) expression

The C++-style casting takes the syntax as

type (expression)

Let us see the concept of type casting in C++ with a small example:

```
#include <iostream.h>
void main()
{
int a;
float b,c;
cout<< "Enter the value of a:";
cin>>a;
cout<< "n Enter the value of b:";
cin>>b;
c = float(a)+b;
cout<<"n The value of c is:"<<c;
}
```

The output of the above program is

```
Enter the value of a: 10
Enter the value of b: 12.5
The value of c is: 22.5
```

In the above program 'a' is declared as integer and b and c are declared as float. In the type conversion statement namely

```
c = float(a)+b;
```

The variable a of type integer is converted into float type and so the value 10 is converted as 10.0 and then is added with the float variable b with value 12.5 giving a resultant float variable c with value as 22.5

Explicit Constructors

The keyword explicit is a Function Specifier." The explicit specifier applies only to constructors. Any time a constructor requires only one argument either of the following can be used to initialize the object. The reason for this is that whenever a constructor is created that takes one argument, it also implicitly creates a conversion from the type of that argument to the type of the class. A constructor specified as explicit will be used only when an initialization uses the normal constructor syntax, Data (x). No automatic conversion will take place and Data = x will not be allowed. Thus, an explicit constructor creates a "nonconverting constructor."

Example:

```
class Data
{
```

```
explicit Data(float x); // Explicit constructor
{ }

};
```

Implicit Constructors

If a constructor is not stated as explicit, then it is by default an implicit constructor.

UNIT 3

Function and class templates - Exception handling – try-catch-throw paradigm – exception specification – terminate and Unexpected functions – Uncaught exception.

Templates

Function templates

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>
myType GetMax (myType a, myType b) {
 return (a>b?a:b);
}
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:

function_name <type> (parameters);

For example, to call GetMax to compare two integer values of type int we can write:

```
int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is the entire example:

```
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

Output

6
10

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.

In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long. The compiler has instantiated and then called each time the appropriate version of the function.

As you can see, the type T is used within the GetMax() template function even to declare new objects of that type:

T result;

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.

In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <long>). So we could have written instead:

```
int i,j;
GetMax (i,j);
```

Since both i and j are of type int, and the compiler can automatically find out that the template parameter can only be int. This implicit method produces exactly the same result:

```
// function template II
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  return (a>b?a:b);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax(i,j);
  n=GetMax(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

Output

6
10

Notice how in this case, we called our function template GetMax() without explicitly specifying the type between angle-brackets <>. The compiler automatically determines what type is needed on each call.

Because our template function includes only one template parameter (class T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i;
long l;
k = GetMax (i,l);
```

This would not be correct, since our GetMax function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U>
T GetMin (T a, U b) {
  return (a<b?a:b);
}
```

In this case, our function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin() with:

```
int i,j;
long l;
i = GetMin<int,long> (j,l);
```

or simply:

```
i = GetMin (j,l);
```

even though j and l have different types, since the compiler can determine the appropriate instantiation anyway.

Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
```

```
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template <...> prefix:

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

Output

100

Notice the syntax of the definition of member function getmax:

```
template <class T>
T mypair<T>::getmax ()
```

Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second T refers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

Template specialization

If we want to define a different implementation for a template when a specific type is passed as template parameter, we can declare a specialization of that template.

For example, let's suppose that we have a very simple class called mycontainer that can store one element of any type and that it has just one member function called increase, which increases its value. But we find that when it stores an element of type char it would be more convenient to have a completely different implementation with a function member uppercase, so we decide to declare a class template specialization for that type:

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
   T element;
  public:
   mycontainer (T arg) {element=arg;}
   T increase () {return ++element;}          8
};                                            J

// class template specialization:
template <>
class mycontainer <char> {
   char element;
  public:
   mycontainer (char arg) {element=arg;}
   char uppercase ()
   {
    if ((element>='a')&&(element<='z'))
```

```
      element+=’A’-‘a’;
      return element;
    }
};

int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar (‘j’);
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

This is the syntax used in the class template specialization:

```
template <> class mycontainer <char> { ... };
```

First of all, notice that we precede the class template name with an emptytemplate<> parameter list. This is to explicitly declare it as a template specialization.

But more important than this prefix, is the <char> specialization parameter after the class template name. This specialization parameter itself identifies the type for which we are going to declare a template class specialization (char). Notice the differences between the generic class template and the specialization:

```
template <class T> class mycontainer { ... };
template <> class mycontainer <char> { ... };
```

The first line is the generic template, and the second one is the specialization.

When we declare specializations for a template class, we must also define all its members, even those exactly equal to the generic template class, because there is no “inheritance” of members from the generic template to the specialization.

Non-type parameters for templates

Besides the template arguments that are preceded by the class or typename keywords , which represent types, templates can also have regular typed parameters, similar to those found in functions. As an example, have a look at this class template that is used to contain sequences of elements:

```
// sequence template
#include <iostream>
using namespace std;
```

```
template <class T, int N>
class mysequence {
   T memblock [N];
  public:
   void setmember (int x, T value);
   T getmember (int x);
};

template <class T, int N>
void mysequence<T,N>::setmember (int x, T value) {
  memblock[x]=value;
}

template <class T, int N>
T mysequence<T,N>::getmember (int x) {
  return memblock[x];
}

int main () {
  mysequence <int,5> myints;
  mysequence <double,5> myfloats;
  myints.setmember (0,100);
  myfloats.setmember (3,3.1416);
  cout << myints.getmember(0) << '\n';
  cout << myfloats.getmember(3) << '\n';
  return 0;
}
```

Output

100

3.1416

It is also possible to set default values or types for class template parameters. For example, if the previous class template definition had been:

```
template <class T=char, int N=10> class mysequence {..};
```

We could create objects using the default template parameters by declaring:

```
mysequence<> myseq;
```

Which would be equivalent to:

```
mysequence<char,10> myseq;
```

Exceptions

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called *handlers*.

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a *try block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

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

int main () {
  try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "An exception occurred. Exception Nr. " << e << endl;
  }
  return 0;
}
```

Output

An exception occurred. Exception Nr. 20

The code under exception handling is enclosed in a try block. In this example this code simply throws an exception:

```
throw 20;
```

A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.

The exception handler is declared with the catch keyword. As you can see, it follows

immediately the closing brace of the try block. The catch format is similar to a regular function that always has at least one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught.

We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed.

If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is. This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last:

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

In this case the last handler would catch any exception thrown with any parameter that is neither an int nor a char.

After an exception has been handled the program execution resumes after the try-catch block, not after the throw statement!.

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

```
try {
  try {
      // code here
  }
  catch (int n) {
      throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

Exception specifications

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a throw suffix to the function declaration:

float myfunction (*char* param) *throw* (*int*);

This declares a function called myfunction which takes one agument of type char and returns an element of type float. The only exception that this function might throw is an exception of type int. If it throws an exception with a different type, either directly or indirectly, it cannot be caught by a regular int-type handler.

If this throw specifier is left empty with no type, this means the function is not allowed to throw exceptions. Functions with no throw specifier (regular functions) are allowed to throw exceptions with any type:

int myfunction (*int* param) *throw*(); *// no exceptions allowed*
int myfunction (*int* param); *// all exceptions allowed*

UNIT 4

C++ Inheritance

What is Inheritance?

Inheritance is the process by which new classes called *derived* classes are created from existing classes called *base* classes. The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class.

For example, a programmer can create a *base* class named fruit and define *derived* classes as mango, orange, banana, etc. Each of these derived classes, (mango, orange, banana, etc.) has all the features of the *base* class (fruit) with additional attributes or features specific to these newly created derived classes. Mango would have its own defined features, orange would have its own defined features, banana would have its own defined features, etc.

This concept of *Inheritance* leads to the concept of *polymorphism.*

Features or Advantages of Inheritance

- ***Reusability:***

  Inheritance helps the code to be reused in many situations. The base class is defined and once it is compiled, it need not be reworked. Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.

- ***Saves Time and Effort:***

  The above concept of reusability achieved by inheritance saves the programmer time and effort, because the main code written can be reused in various situations as needed.

- ***Increases Program Structure which results in greater reliability.***
- ***Polymorphism***

General Format for implementing the concept of Inheritance:

class derived_classname: access specifier baseclassname

For example, if the *base* class is *exforsys* and the derived class is sample it is specified as:

```
class sample: public exforsys
```

The above makes sample have access to both *public* and *protected* variables of base class *exforsys*. Reminder about public, private and protected access specifiers:

- If a member or variables defined in a class is private, then they are accessible by members of the same class only and cannot be accessed from outside the class.
.
- Public members and variables are accessible from outside the class.
.
- Protected access specifier is a stage between private and public. If a member functions or variables defined in a class are protected, then they cannot be accessed from outside the class but can be accessed from the derived class.

Inheritance Example:

```
class exforsys
{
private:
int x;

public:
exforsys(void) { x=0; }
void f(int n1)
{
x= n1*5;
}

void output(void) { cout<<x;
}
};

class sample: public exforsys
{
public:
sample(void) { s1=0; }

void f1(int n1)
{
s1=n1*10;
}

void output(void)
{
```

```
exforsys::output();
cout << s1;
}

private:
int s1;
};

int main(void)
{
sample s;
s.f(10);
s.output();
s.f1(20);
s.output();
}
```

The output of the above program is

```
50
200
```

In the above example, the derived class is sample and the base class is *exforsys*. The *derived* class defined above has access to all *public* and *private* variables. *Derived* classes cannot have access to base class *constructors* and *destructors*. The derived class would be able to add new member functions, or variables, or new constructors or new destructors. In the above example, the derived class sample has new member function f1( ) added in it. The line:

```
sample s;
```

creates a derived class object named as s. When this is created, space is allocated for the data members inherited from the base class *exforsys* and space is additionally allocated for the data members defined in the derived class *sample*.

The *base* class constructor *exforsys* is used to initialize the base class data members and the *derived* class *constructor* sample is used to initialize the data members defined in *derived* class.

The access specifier specified in the line:

class sample: public exforsys

Public indicates that the *public* data members which are inherited from the *base* class by the derived class sample remains *public* in the *derived* class.

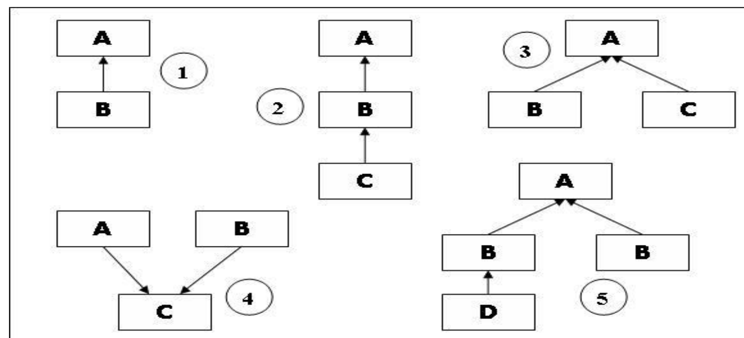
A derived class inherits every member of a base class except:

- its constructor and its destructor
- its friends
- its operator=() members

Types of Inheritance

There are five different inheritances supported in C++:

(1) Simple / Single
(2) Multilevel
(3) Hierarchical
(4) Multiple
(5) Hybrid



Accessibility modes and Inheritance

We can use the following chart for seeing the accessibility of the members in the Base class (first class) and derived class (second class).

		Inheritance Mode		
		public	protected	private
Members in Base Class	public	public	protected	private
	protected	protected	protected	private
	private	X	X	X
		Members in derived class		

Here X indicates that the members are not inherited, i.e. they are not accessible in the derived class.

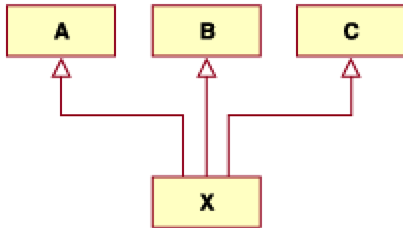
Multiple Inheritance

We can derive a class from any number of base classes. Deriving a class from more than one direct base class is called multiple inheritance.

In the following example, classes A, B, and C are direct base classes for the derived class X:

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class X : public A, private B, public C { /* ... */ };
```

The following inheritance graph describes the inheritance relationships of the above example. An arrow points to the direct base class of the class at the tail of the arrow:

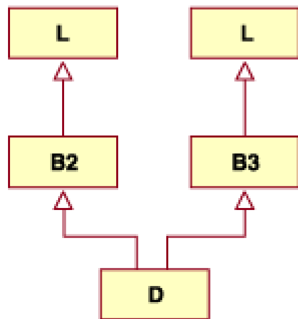


The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors.

A direct base class cannot appear in the base list of a derived class more than once:

```
class B1 { /* ... */ };                   // direct base class
class D : public B1, private B1 { /* ... */ }; // error
```

However, a derived class can inherit an indirect base class more than once, as shown in the following example:




```
class L { /* ... */ };                  // indirect base class
class B2 : public L { /* ... */ };
class B3 : public L { /* ... */ };
class D : public B2, public B3 { /* ... */ }; // valid
```

In the above example, class D inherits the indirect base class L once through class B2 and once through class B3. However, this may lead to ambiguities because two subobjects of class L exist, and both are accessible through class D. You can avoid this ambiguity by referring to class L using a qualified class name. For example:

B2::L

or

B3::L.

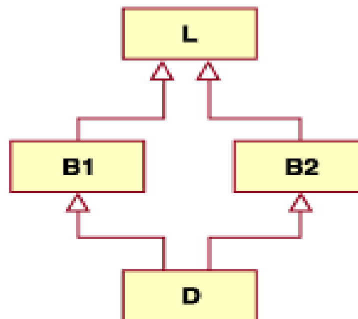
we can also avoid this ambiguity by using the base specifier virtual to declare a base class.

Virtual Base Classes

Suppose you have two derived classes B and C that have a common base class A, and you also have another class D that inherits from B and C. You can declare the base class A as virtual to ensure that B and C share the same subobject of A.

In the following example, an object of class D has two distinct subobjects of class L, one through class B1 and another through class B2. You can use the keyword virtual in front of the base class specifiers in the base lists of classes B1 and B2 to indicate that only one subobject of type L, shared by class B1 and class B2, exists.

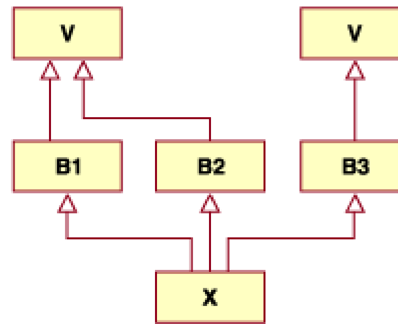
For example:




```
class L { /* ... */ }; // indirect base class
class B1 : virtual public L { /* ... */ };
class B2 : virtual public L { /* ... */ };
class D : public B1, public B2 { /* ... */ }; // valid
```

Using the keyword virtual in this example ensures that an object of class D inherits only one subobject of class L.

A derived class can have both virtual and nonvirtual base classes. For example:




```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 : virtual public V { /* ... */ };
class B3 : public V { /* ... */ };
class X : public B1, public B2, public B3 { /* ... */
};
```

In the above example, class X has two subobjects of class V, one that is shared by classes B1 and B2 and one through class B3.

Abstract Classes

An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. You can declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

The following is an example of an abstract class:

```
class AB {
public:
virtual void f() = 0;
};
```

Function AB::f is a pure virtual function. A function declaration cannot have both a pure specifier and a definition. For example, the compiler will not allow the following:

```
class A {
virtual void g() { } = 0;
};
```

You cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion, nor can you declare an object of an abstract class. You can, however, declare pointers and references to an abstract class. The following example demonstrates this:

```
class A {
virtual void f() = 0;
```

```
};

class A {
virtual void f() { }
};

// Error:
// Class A is an abstract class
// A g();

// Error:
// Class A is an abstract class
// void h(A);
A& i(A&);

int main() {
// Error:
// Class A is an abstract class
//   A a;

   A* pa;
   B b;

// Error:
// Class A is an abstract class
//   static_cast<A>(b);
}
```

Class A is an abstract class. The compiler would not allow the function declarations A g() or void h(A), declaration of object a, nor the static cast of b to type A.

Virtual member functions are inherited. A class derived from an abstract base class will also be abstract unless you override each pure virtual function in the derived class.

For example:

```
class AB {
public:
  virtual void f() = 0;
};

class D2 : public AB {
  void g();
};

int main() {
```

```
  D2 d;
}
```

The compiler will not allow the declaration of object d because D2 is an abstract class; it inherited the pure virtual function f()from AB. The compiler will allow the declaration of object d if you define function D2::g().

Note that you can derive an abstract class from a nonabstract class, and you can override a non-pure virtual function with a pure virtual function.

You can call member functions from a constructor or destructor of an abstract class. However, the results of calling (directly or indirectly) a pure virtual function from its constructor are undefined. The following example demonstrates this:

```
clss A {
  A() {
    direct();
    indirect();
  }
  virtual void direct() = 0;
  virtual void indirect() { direct(); }
};
```

The default constructor of A calls the pure virtual function direct() both directly and indirectly (through indirect()).

The compiler issues a warning for the direct call to the pure virtual function, but not for the indirect call.

Polymorphism

Polymorphism is the phenomenon where the same message sent to two different objects produces two different set of actions. Polymorphism is broadly divided into two parts:

- *Static polymorphism* – exhibited by overloaded functions.
- *Dynamic polymorphism* – exhibited by using late binding.

Static Polymorphism

Static polymorphism refers to an entity existing in different physical forms simultaneously. Static polymorphism involves binding of functions based on the number, type, and sequence of arguments. The various types of parameters are specified in the function declaration, and therefore the function can be bound to calls at compile time. This form of association is called *early binding*. The term *early binding* stems from the fact that when the program is executed, the calls are already bound to the appropriate functions.

The resolution of a function call is based on number, type, and sequence of arguments declared for each form of the function. Consider the following function declaration:

```
void add(int , int);
void add(float, float);
```

When the *add()* function is invoked, the parameters passed to it will determine which version of the function will be executed. This resolution is done at compile time.

Dynamic Polymorphism

Dynamic polymorphism refers to an entity changing its form depending on the circumstances. A function is said to exhibit dynamic polymorphism when it exists in more than one form, and calls to its various forms are resolved dynamically when the program is executed. The term *late binding* refers to the resolution of the functions at run-time instead of compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context.

Static Vs Dynamic Polymorphism

- Static polymorphism is considered more efficient, and dynamic polymorphism more flexible.
- Statically bound methods are those methods that are bound to their calls at compile time. Dynamic function calls are bound to the functions during run-time. This involves the additional step of searching the functions during run-time. On the other hand, no run-time search is required for statically bound functions.
- As applications are becoming larger and more complicated, the need for flexibility is increasing rapidly. Most users have to periodically upgrade their software, and this could become a very tedious task if static polymorphism is applied. This is because any change in requirements requires a major modification in the code. In the case of dynamic binding, the function calls are resolved at run-time, thereby giving the user the flexibility to alter the call without having to modify the code.
- To the programmer, efficiency and performance would probably be a primary concern, but to the user, flexibility or maintainability may be much more important. The decision is thus a trade-off between efficiency and flexibility.

Introduction To Virtual Functions

Polymorphism, one of the three main attributes of an OOP language, denotes a process by which different implementations of a function can be accessed by the use of a single name. Polymorphism also means "one interface, multiple methods."

C++ supports polymorphism both at run-time and at compile-time. The use of overloaded functions is an example of compile-time polymorphism. Run-time polymorphism can be achieved by the use of both derived classes and *virtual functions.*

Pointers to Derived Types

We know that pointer of one type may not point to an object of another type. You'll now learn about the one exception to this general rule: a pointer to an object of a base class can also point to any object derived from that base class.

Similarly, a reference to a base class can also reference any object derived from the original base class. In other words, a base class reference parameter can receive an object of types derived from the base class, as well as objects within the base class itself.

Virtual Functions

How does C++ handle these multiple versions of a function? Based on the parameters being passed, the program determines at run-time which version of the virtual function should be the recipient of the reference. It is the type of object being pointed to, not the type of pointer, that determines which version of the virtual function will be executed!

To make a function *virtual,* the virtual keyword must precede the function declaration in the base class. The redefinition of the function in any derived class does not require a second use of the virtual keyword. Have a look at the following sample program to see how this works:

```
#include <iostream.h>
using namespace std;
class bclass {
public:
virtual void whichone() {
cout << "bclass\n";
    }
};
class dclass1 : public bclass {
public:
void whichone() {
cout << "dclass1\n";
    }
};
class dclass2 : public bclass {
public:
void whichone() {
cout << "dclass2\n";
    }
};
int main()
{
bclass Obclass;
```

```
bclass *p;
dclass1 Odclass1;
dclass2 Odclass2;
// point to bclass
p = &Obclass;
// access bclass's whichone()
p->whichone();
// point to dclass1
p = &Odclass1;
// access dclass1's whichone()
p->whichone();
// point to dclass2
p = &Odclass2;
// access dclass2's whichone()
p->whichone();
return 0;
}
```

The output from this program looks like this:

```
bclass
dclass1
dclass2
```

Notice how the type of the object being pointed to, not the type of the pointer itself, determines which version of the virtual whichone() function is executed.

Virtual Functions and Inheritance

. Virtual functions are inherited intact by all subsequently derived classes, even if the function is not redefined within the derived class. So, if a pointer to an object of a derived class type calls a specific function, the version found in its base class will be invoked. Look at the modification of the above program. Notice that the program does not define the whichone() function in d class2.

```
#include <iostream.h>
using namespace std;
class bclass {
public:
virtual void whichone() {
cout << "bclass\n";
    }
};
class dclass1 : public bclass {
public:
```

```
void whichone() {
cout << “dclass1\n”;
    }
};
class dclass2 : public bclass {
};
int main()
{
bclass Obclass;
bclass *p;
dclass1 Odclass1;
dclass2 Odclass2;
p = &Obclass;
p->whichone();
p = &Odclass1;
p->whichone();
p = &Odclass2;
// accesses dclass1’s function
p->whichone();
return 0;
}
```

The output from this program looks like this:

```
bclass
dclass1
bclass
```

Rtti Constituents

The operators typeid and dynamic_cast<> offer two complementary forms of accessing the runtime type information of their operands. The operand’s runtime type information itself is stored in a type_info object.

It’s important to realize that RTTI is applicable solely to polymorphic objects; a class must have at least one virtual-member function in order to have RTTI support for its objects.

std::type_info

For every distinct type, C++ instantiates a corresponding std::type_info (defined in <typeinfo>) object. The interface is as follows:

```
namespace std {
class type_info
 {
```

```
public:
virtual ~type_info(); //type_info can serve as a base class
// enable comparison
bool operator==(const type_info& rhs ) const;
// return !( *this == rhs)
bool operator!=(const type_info& rhs ) const;
bool before(const type_info& rhs ) const; // ordering
//return a C-string containing the type's name
const char* name() const;
private:
//objects of this type cannot be copied
type_info(const type_info& rhs );
type_info& operator=(const type_info& rhs);
}; //type_info
}
```

All objects of the same class share a single type_info object. The most widely used member functions of type_info are name() and operator==. But before you can invoke these member functions, you have to access the type_info object itself. How? Operator typeid takes either an object or a type name as its argument and returns a matching type_info object. The dynamic type of an object can be examined as follows:

```
OnRightClick (File & file)
{
if ( typeid( file) == typeid( TextFile ) )
 {
//received a TextFile object; printing should be enabled
 }
else
 {
//not a TextFile object; printing disabled
 }
}
```

To understand how it works, look at the highlighted source line:

if (typeid(file) == typeid(TextFile)).

The if statement tests whether the dynamic type of file is TextFile (the static type of file is File, of course). The leftmost expression, typeid(file), returns a type_info object that holds the necessary runtime type information associated with the object file. The rightmost expression, typeid(TextFile), returns the type information associated with class TextFile. (When typeid is applied to a class name rather than an object, it always returns a type_info object that corresponds to that class name.)

As shown earlier, type_info overloads the operator ==. Therefore, the type_info object returned by the leftmost typeid expression is compared to the type_info object returned by the rightmost typeid expression. If file is an instance of TextFile, the if statement evaluates to true. In that case, OnRightClick should display an additional option in the menu: print(). On the other hand, if file is not a TextFile, the if statement evaluates to false, and the print() option is disabled.

dynamic_cast<>

OnRightClick() doesn't really need to know whether file is an instance of class TextFile (or of any other class, for that matter). Rather, all it needs to know is whether file *is-a* TextFile. An object *is-a* TextFile if it's an instance of class TextFile or any class derived from it. For this purpose, you use the operator dynamic_cast<>. dynamic_cast<> takes two arguments: a type name, and an object that dynamic_cast<> attempts to cast at runtime. For example:

```
//attempt to cast file to a reference to
//an object of type TextFile
dynamic_cast <TextFile &> (file);
```

If the attempted cast succeeds, the second argument *is-a* TextFile. But how do you know whether dynamic_cast<> was successful?

There are two flavors of dynamic_cast<>; one uses pointers and the other uses references. Accordingly, dynamic_cast<> returns a pointer or a reference of the desired type when it succeeds. When dynamic_cast<> cannot perform the cast, it returns NULL; or, in the case of a reference, it throws an exception of type std::bad_cast:

```
TextFile * pTest = dynamic_cast <TextFile *>
(&file); //attempt to cast
//file address to a pointer to TextFile
if (pTest) //dynamic_cast succeeded, file is-a TextFile
{
//use pTest
}
else // file is not a TextFile; pTest has a NULL value
{
}
```

Remember to place a reference dynamic_cast<> expression inside a try block and include a suitable catch statement to handle std::bad_cast exceptions.

Now you can revise OnRightClick() to handle HTMLFile objects properly:

```
OnRightClick (File & file)
{
```

```
try
 {
TextFile temp = dynamic_cast<TextFile&> (file);
//display options, including "print"
switch (message)
  {
case m_open:
temp.open(); //either TextFile::open or HTMLFile::open
break;
case m_print:
temp.print();//either TextFile::print or HTMLFile::print
break;
}//switch
}//try
catch (std::bad_cast& noTextFile)
 {
// treat file as a BinaryFile; exclude"print"
 }
}// OnRightClick
```

The revised version of OnRightClick() handles an object of type HTMLFile properly. When the user clicks the open option in the file manager application, OnRightClick() invokes the member function open() of its argument. Likewise, when it detects that its argument is a TextFile, it displays a print option.

This hypothetical file manager example is a bit contrived; with the use of templates and more sophisticated design, it could have been implemented without dynamic_cast. Yet dynamic type casts have other valid uses in C++, as I'll show next.

Cross Casts

A *cross cast* converts a multiply-inherited object to one of its secondary base classes. To see what a cross cast does, consider the following class hierarchy:

```
struct A
{
int i;
virtual ~A () {} //enforce polymorphism; needed for dynamic_cast
};
struct B
{
bool b;
};

struct D: public A, public B
{
```

```
int k;
D() { b = true; i = k = 0; }
};

A *pa = new D;
B *pb = dynamic_cast<B*> pa; //cross cast; access the second base
//of a multiply-derived object
```

The static type of pa is “A *”, whereas its dynamic type is “D *”. A simple static_cast<> cannot convert a pointer to A into a pointer to B, because A and B are unrelated. Don’t even think of a brute force cast. It will cause disastrous results at runtime because the compiler will simply assign pa to pb; whereas the B sub-object is located at a different address within D than the A sub-object. To perform the cross cast properly, the value of pb has to be calculated at runtime. After all, the cross cast can be done in a source file that doesn’t even know that class D exists! The following listing demonstrates why a dynamic cast, rather than a compile-time cast, is required in this case:

```
A *pa = new D;
// disastrous; pb points to the sub-object A within d
B pb = (B) pa; bool bb = pb->b; // bb has an undefined value
// pb was not properly
// adjusted; pa and pb are identical
cout<< “pa: “ << pa << “ pb: “<<pb <<endl;
pb = dynamic_cast<B*> (pa); //cross cast; adjust pb correctly
bb= pb->b; //OK, bb is true
// OK, pb was properly adjusted;
// pa and pb have distinct values
cout<< “pa: “<< pa << “ pb: “ << pb <<endl;
```

The code displays two lines of output; the first shows that the memory addresses of pa and pb are identical. The second line shows that the memory addresses of pa and pb are indeed different after performing a dynamic cast as needed.

Downcasting from a Virtual Base

A *downcast* is a cast from a base to a derived object. Before the advent of RTTI, downcasts were regarded as bad programming practice and notoriously unsafe. dynamic_cast<> enables you to use safe and simple downcasts from a virtual base to its derived object. Look at the following example:

```
struct V
{
virtual ~V (){} //ensure polymorphism
};
struct A: virtual V {};
```

```
struct B: virtual V {};
struct D: A, B {};

#include <iostream>
using namespace std;
int main()
{
V *pv = new D;
A* pa = dynamic_cast<A*> (pv); // downcast
cout<< “pv: “<< pv << “ pa: “ << pa <<endl; // OK, pv and pa have
//different addresses
}
```

V is a virtual base for classes A and B. D is multiply-inherited from A and B. Inside main(), pv is declared as a “V *” and its dynamic type is “D *”. Here again, as in the cross-cast example, the dynamic type of pv is needed in order to properly downcast it to a pointer to A. A static_cast<> would be rejected by the compiler in this case, as the memory layout of a virtual sub-object might be different from that of a non-virtual sub-object. Consequently, it’s impossible to calculate at compile time the address of the sub-object A within the object pointed to by pv. As the output of the program shows, pv and pa indeed point to different memory addresses.

UNIT 5

C++ Standard Input Output Stream

C++ programming language uses the concept of streams to perform input and output operations using the keyboard and to display information on the monitor of the computer.

What is a Stream?

A stream is an object where a program can either insert or extract characters to or from it. The standard input and output stream objects of C++ are declared in the header file *iostream*.

Standard Input Stream

- Generally, the device used for input is the keyboard. For inputting, the keyword *cin* is used, which is an object.
- The overloaded operator of extraction, >>, is used on the standard input stream, in this case: *cin* stream.

Syntax for using the standard input stream is cin *followed by the operator* >> *followed by the* variable *that stores the data extracted from the stream.*

For example:

```
int prog;
cin >> prog;
```

In the example above, the variable *prog* is declared as an integer type variable. The next statement is the *cin* statement. The *cin* statement waits for input from the user's keyboard that is then stored in the integer variable *prog*.

The input stream *cin* wait before proceeding for processing or storing the value. This duration is dependent on the user pressing the RETURN key on the keyboard. The input stream *cin* waits for the user to press the RETURN key then begins to process the command. It is also possible to request input for more than one variable in a single input stream statement. A single *cin* statement is as follows:

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

is the same as:

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

In both of the above cases, two values are input by the user, one value for the variable x and another value for the variable y.

```
// This is a sample program        This is a comment Statement

#include <iostream.h>               Header File Inclusion Statement
void main()
{
int sample, example;
cin >> sample;
cin >> example;
}
```

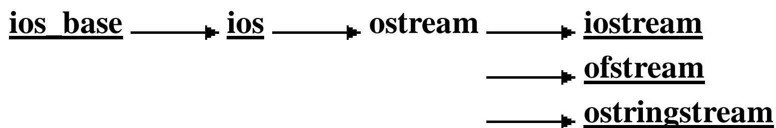
In the above example, two integer variables are input with values.

- The programmer can produce input of any data type.
- It is also possible to input strings in C++ program using *cin*. This is performed using the same procedures.
- The vital point to note is *cin* stops when it encounters a blank space.
- When using a cin, it is possible to produce only one word.

If a user wants to input a sentence, then the above approach would be tiresome. For this purpose, there is a function in C++ called *getline*.

Ostream

Output Stream



ostream objects are stream objects used to write and format output as sequences of characters. Specific members are provided to perform these output operations, which can be divided in two main groups:

- Formatted output
    - These member functions interpret and format the data to be written as sequences of characters.
    - These type of operation is performed using member and global functions that overload the insertion operator
- Unformatted output

- Most of the other member functions of the ostream class are used to perform unformatted output operations, i.e. output operations that write the data as it is, with no formatting adaptations.
- These member functions can write a determined number of characters to the output character sequence (put, write) and manipulate the *put pointer* (seekp, tellp).

Additionaly, a member function exists to synchronize the stream with the associated external destination of characters: sync.

The standard objects cout, cerr and clog are instantiations of this class.

The class inherits all the internal fields from its parent classes ios_base and ios:

Formatting information

- **format flags:** a set of internal indicators describing how certain input/output operations shall be interpreted or generated. The state of these indicators can be obtained or modified by calling the members flags, *setf* and *unsetf,* or by using manipulators.
- **field width:** describes the width of the next element to be output. This value can be obtained/modified by calling the member function width or parameterized manipulator *setw*.
- **display precision:** describes the decimal precision to be used to output floating-point values. This value can be obtained / modified by calling member precision or parameterized manipulator *setprecision*.
- **fill character:** character used to pad a field up to the *field width*. It can be obtained or modified by calling member function fill or parameterized manipulator *setfill*.
- **locale object:** describes the localization properties to be considered when formatting i/o operations. The locale object used can be obtained calling member *getloc* and modified using member imbue.

State information

- **error state:** internal indicator reflecting the integrity and current error state of the stream. The current object may be obtained by calling *rdstate* and can be modified by calling *clear* and *setstate*. Individual values may be obtained by calling *good*, *eof*, *fail* and *bad*.
- **exception mask:** internal exception status indicator. Its value can be retrieved/modified by calling member *exceptions*.

Formatted output:

operator<< Insert data with format (public member function)

Unformatted output:

put Put character (public member function)
write Write block of data (public member function)

Positioning:

tellp Get position of put pointer (public member function)
seekp Set position of put pointer (public member function)

Synchronization:

flush Flush output stream buffer (public member function)

Prefix/Suffix:

sentry Perform exception safe prefix/suffix operations (public member classes)

I/O Manipulators

What is a Manipulator?

Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer's choice of display.

There are numerous manipulators available in C++. Some of the more commonly used manipulators are provided here below:

endl Manipulator:

- This manipulator has the same functionality as the '\n' newline character.

For example:

```
cout << "Exforsys" << endl;
cout << "Training";
```

produces the output:

```
Exforsys
Training
```

setw Manipulator:

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

setw(x)

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

```
#include <iostream.h>
#include <iomanip.h>


void main( )
{
int x1=12345,x2= 23456, x3=7892;
cout << setw(8) << ”Exforsys” << setw(20) <<
”Values” << endl
<< setw(8) << “E1234567” << setw(20)<< x1 << end
<< setw(8) << “S1234567” << setw(20)<< x2 << end
<< setw(8) << “A1234567” << setw(20)<< x3 <<
end;
}
```

The output of the above example is:

```
setw(8)   setw(20)
Exforsys Values
E1234567 12345
S1234567 23456
A1234567 7892
```

setfill Manipulator:

This is used after setw manipulator. If a value does not entirely fill a field, then the character specified in the setfill argument of the manipulator is used for filling the fields.

```
#include <iostream.h>
#include <iomanip.h>
void main( )
{
cout << setw(10) << setfill(‘$’) << 50 << 33 << endl;
}
```

The output of the above program is

$$$$$$$$5033

This is because the setw sets 10 width for the field and the number 50 has only 2 positions in it. So the remaining 8 positions are filled with $ symbol which is specified in the setfill argument.

setprecision Manipulator:

The setprecision Manipulator is used with floating point numbers. It is used to set the number of digits printed to the right of the decimal point. This may be used in two forms:

- fixed
- scientific

These two forms are used when the keywords fixed or scientific are appropriately used before the setprecision manipulator.

- The keyword fixed before the setprecision manipulator prints the floating point number in fixed notation.
- The keyword scientific before the setprecision manipulator prints the floating point number in scientific notation.

```
#include <iostream.h>
#include <iomanip.h>
void main( )
{
float x = 0.1;
cout << fixed << setprecision(3) << x << endl;
cout << sceintific << x << endl;
}
```

The above gives ouput as:

0.100
1.000000e-001

The first cout statement contains fixed notation and the setprecision contains argument 3. This means that three digits after the decimal point and in fixed notation will output the first cout statement as 0.100. The second cout produces the output in scientific notation. The default value is used since no setprecision value is provided.

File I/O with Streams

Many real-life problems handle large volumes of data, therefore we need to use some devices such as floppy disk or hard disk to store the data.

The data is stored in these devices using the concept of files. A file is a collection of related data stored in a particular area on the disk.

Programs can be designed to perform the read and write operations on these files.

A program typically involves either or both of the following kinds of data communication:

- Data transfer between the console unit and the program.
- Data transfer between the program and a disk file.
- The input/output system of C++ handles file operations which are very much similar to the console input and output operations.

It uses file streams as an interface between the programs and the files.

The stream that supplies data to the program is known as input stream and the one that receives data from the program is known as output stream.

In other words, the input stream extracts or reads data from the file and the output stream inserts or writes data to the file.

Classes for the file stream operations
The I/O system of C++ contains a set of classes that define the file handling methods.

These include ifstream, ofstream and fstream.

These classes, designed to manage the disk files, are declared in fstream.h and therefore we must include this file in any program that uses files.

Details of some useful classes :
filebuf
Its purpose is to set the file buffer to read and write. Contains openprot constant used in the open() of the filestream classes. Also contains close() and open() as member functions.
fstreambase
Provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream classes. Contains open() and close() functions.

ifstream

Provides input operations. Contains open() with default input mode. Inherits the functions get(), getline(), read(), seekg() and tellg() functions from istream.

ofstream
Provides output operations. Contains open() with default output mode. Inherits put(), seekp(), tellp(), and write() functions from ostream.

fstream
Provides support for simultaneous input and output operations. Contains open() with default input mode. Inherits all the functions from istream and ostream classes through iostream.

The ifstream, ofstream and fstream classes are declared in the file fstream.h
The istream and ostream classes are also included in the fstream.h file.

Opening and closing a file
For opening a file, we must first create a file stream and than link it to the filename.

A filestream can be defined using the classes ifstream, ofstream, and fstream that are contained in the header file fstream.h

A file can be opened in two ways:

- Using the constructor function of the class. This method is useful when we open only one file in the stream.
- Using the member function open() of the class. This method is used when we want to manage multiple files using one stream.

Using Constructor
Create a file stream object to manage the stream using the appropriate class. That is, the class ofstream is used to create the output stream and the class ifstream to create the input stream.

Initialize the file object with the desired filename, e.g.:

ofstream outfile("sample.txt");

The above statement creates an object outfile of class ofstream that manages the output stream. This statement also opens the file sample.txt and attaches it to the output stream for writing.
Similarly, the statement declared in as an ifstream object and attaches to the file "sample.txt" for reading.

ifstream infile("sample.txt");

Program: Writing and reading data into file, using constructors

```
# include <fstream.h>
void main()
{
ofstream outfile("sample.txt"); // create file for output
char ch = 'a';
int i = 12;
float f = 4356.15;
char arr[ ] = "hello";
outfile << ch << endl <<< endl << f << endl << arr; //send the data to file
outfile.close();

ifstream infile("sample.txt");


infile >> ch >> i >> f >> arr; // read data from file

cout << ch << i << f << arr; // send data to screen
}
```

To write data into the file, character by character.

```
#include<fstream.h>
#include<string.h>
void main()
{
char str[]="C++ is superset of C. It is an object-oriented /
programming language.";

ofstream outfile("sample2.txt"); // Open the file in write mode

for(int i = 0; i < strlen(str); i++)
outfile.put(str[i]); // write data into the file, character by character.
}
```

Writing and reading Objects of a class :
So far we have done I/O of basic data types. Since the class objects are the central elements of C++ programming, it is quite natural that the language supports features for writing and reading from the disk files objects directly.

The binary input and output functions read() and write() are designed to do exactly this job.

The write() function is used to write the object of a class into the specified file and

read() function is used to read the object of the class from the file.

Both these functions take two arguments:
1. address of object to be written.
2. size of the object.

The address of the object must be cast to the type pointer to char.

One important point to remember is that only data members are written to the disk file and the member functions are not.

Writing an object into the file

```
#include<fstream.h>
class Person
{
private:
char name[40];
int age;
public:
void getData()
{
cout << "\n Enter name:"; cin >> name;
cout << "\n Enter age:"; cin >> age;
}
} ; // End of the class definition

void main()
{
Person per ; // Define an object of Person class

per.getData(); // Enter the values to the data members of the class.

ofstream outfile("Person.txt"); // Open the file in output mode

outfile.write((char*)&per, sizeof(per)); // Write the object into the file
}
```

fstream object can be used for both input & output.

In the open() function we include several mode bits to specify certain aspects of the file object.
app -> To preserve whatever was in the file before. Whatever we write to the file will be appended to the existing contents.

We use in and out because we want to perform both input and output on the file.

eof() is a member function of ios class. It returns a nonzero value if EOF is encountered and a zero otherwise.

Parameters of open() function
ios::app Append to end of the file
ios::ate Go to end of the file on opening
ios::in Open file for reading only
ios::nocreate Open fails if the file does not exist
ios::noreplace Open fails if the file already exists
ios::out Open file for writing only
ios::trunc Delete contents of the file if it exists

File pointers and their manipulations

Each file has two associated pointers known as the file pointers.
One of them is called the **input pointer or get pointer**.
Other is called the **output pointer or put pointer.**

We can use these pointers to move through the files while reading or writing.

The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location.

Functions for manipulation of file pointers
seekg() Moves get pointer (input) to a specified location.
seekp() Moves put pointer (output) to a specified location.
tellg() Gives the current position of the get pointer.
tellp() Gives the current position of the put pointer.

```
infile.seekg(10);
```

Moves the file pointer to the byte number 10. The bytes in a file are numbered beginning from zero.Thus, the pointer will be pointing to the 11th byte in the file.

Object Serialization

Object serialization consists of saving the values that are part of an object, mostly the value gotten from declaring a variable of a class. AT the current standard, C++ doesn't inherently support object serialization. To perform this type of operation, you can use a technique known as binary serialization.

Namespaces

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in “sub-scopes”, each one with its own name.

The format of namespaces is:

namespace identifier
{
entities
}

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:

```
namespace myNamespace
{
int a, b;
}
```

In this case, the variables a and b are normal variables declared within a namespace called myNamespace. In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::. For example, to access the previous variables from outside myNamespace we can write:

```
myNamespace::a
myNamespace::b
```

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

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

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
```

```
  cout << second::var << endl;
  return 0;
}
```

Output:
5
3.1416

In this case, there are two global variables with the same name: var. One is defined within the namespace first and the other one in second. No redefinition errors happen.

using

The keyword using is used to introduce a name from a namespace into the current declarative region. For example:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

Output:
5
2.7183
10
3.1416

Notice how in this code, x (without any name qualifier) refers to first::x whereas y refers to second::y, exactly as our using declarations have specified. We still have access to first::y and second::x using their fully qualified names.

The keyword using can also be used as a directive to introduce an entire namespace:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << endl;
  cout << y << endl;
  cout << second::x << endl;
  cout << second::y << endl;
  return 0;
}
```

Output:

5
10
3.1416
2.7183
In this case, since we have declared that we were using namespace first, all direct uses of x and y without name qualifiers where referring to their declarations in namespace first.

using and using namespace have validity only in the same block in which they are stated or in the entire code if they are used directly in the global scope. For example, if we had the intention to first use the objects of one namespace and then those of another one, we could do something like:

```
// using namespace example
#include <iostream>
```

```
using namespace std;

namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << endl;
  }
  {
    using namespace second;
    cout << x << endl;
  }
  return 0;
}
```

Output:

5
3.1416

Namespace alias

We can declare alternate names for existing namespaces according to the following format:

namespace new_name = current_name;

Namespace std

All the files in the C++ standard library declare all of its entities within the std namespace. That is why we have generally included the using namespace std; statement in all programs that used any entity defined in iostream.

Advantages:

- A namespace defines a new scope.
- Members of a namespace are said to have namespace scope.

- They provide a way to avoid name collisions (of variables, types, classes or functions) without some of the restrictions imposed by the use of classes, and without the inconvenience of handling nested classes

Standard Template Library

The Standard Template Libraries (STL's) are a set of C++ template classes to provide common programming data structures and functions such as doubly linked lists (list), paired arrays (map), expandable arrays (vector), large string storage and manipulation (rope), etc.

STL can be categorized into the following groupings:

- Container classes:
  - Sequences:
    - **vector**: (this tutorial) Dynamic array of variables, struct or objects. Insert data at the end.
      (also see the YoLinux.com tutorial on using and STL list and boost ptr_list to manage pointers.)
    - **deque**: Array which supports insertion/removal of elements at beginning or end of array
    - **list**: (this tutorial) Linked list of variables, struct or objects. Insert/remove anywhere.
  - Associative Containers:
    - **set** (duplicate data not allowed in set), **multiset** (duplication allowed): Collection of ordered data in a balanced binary tree structure. Fast search.
    - **map** (unique keys), **multimap** (duplicate keys allowed): Associative key-value pair held in balanced binary tree structure.
  - Container adapters:
    - **stack** LIFO
    - **queue** FIFO
    - **priority_queue** returns element with highest priority.
  - String:
    - **string**: Character strings and manipulation
    - **rope**: String storage and manipulation
  - **bitset**: Contains a more intuitive method of storing and manipulating bits.
- Operations/Utilities:
  - **iterator**: (examples in this tutorial) STL class to represent position in an STL container. An iterator is declared to be associated with a single container class type.
  - **algorithm**: Routines to find, count, sort, search, ... elements in container classes
  - **auto_ptr**: Class to manage memory pointers and avoid memory leaks.

ANSI String Class

The ANSI string class implements a first-class character string data type that avoids many problems associated with simple character arrays ("C-style strings"). You can define a string object very simply, as shown in the following example:

```
#include <string>
using namespace std;
...
string first_name = "Bjarne";
string last_name;

last_name = "Stroustrup";

string names = first_name + " " + last_name;
cout << names << endl;

names = last_name + ", " + first_name;
cout << names << endl;
```

Member functions

The string class defines many member functions. A few of the basic ones are described below:

<table>
<tr>
<td>Initialization (constructor)</td>
<td>A string object may defined without an initializing value, in which case its initial value is an empty string (zero length, no characters):
string str1;

A string object may also be initialized with

• a string expression:
• string str2 = str1;
• string str3 = str1 + str2;
string str4 (str2); // Alternate form

• a character string literal:
• string str4 = "Hello there";
string str5 ("Goodbye"); // Alternate form

• a single character
Unfortunately, the expected methods don't work:
• string str6 = 'A'; // Incorrect
string str7 ('A'); // Also incorrect</td>
</tr>
</table>

	Instead, we must use a special form with two values: string str7 (1,'A'); // Correct The two values are the desired length of the string and a character to fill the string with. In this case, we are asking for a string of length one, filled with the character **A**. • a substring of another string object: • string str8 = "ABCDEFGHIJKL"; • // Initialize str9 as "CDEFG" • // Starts at character 2 ('C') • // with a length of 5 • // (or the rest of the string, if shorter) string str9 (str8,2,5);

length
 	
Size
	size_type length() const;
	size_type size() const;
	Both of these functions return the length (number of characters) of the string. The **size_type** return type is an unsigned integral type. (The type name usually must be scoped, as in **string::size_type**.)
	string str = "Hello";
	string::size_type len;
	len = str.length(); // len == 5
	len = str.size(); // len == 5

c_str	const char* c_str() const;
	For compatibility with "older" code, including some C++ library routines, it is sometimes necessary to convert a string object into a pointer to a null-terminated character array ("C-style string"). This function does the conversion. For example, you might open a file stream with a user-specified file name:
	string filename;
	cout << "Enter file name: ";
	cin >> filename;
	ofstream outfile (filename.c_str());
	outfile << "Data" << endl;

insert	string& insert(size_type pos, const string& str);
	Inserts a string into the current string, starting at the specified position.
	string str11 = "abcdefghi";
	string str12 = "0123";
	str11.insert (3,str12);
	cout << str11 << endl; // "abc0123defghi"

<table>
<tr><td></td><td>str12.insert (1,”XYZ”);
cout << str12 << endl; // “0XYZ123”</td></tr>
<tr><td>Erase</td><td>string& erase(size_type pos=0, size_type n=npos);
Delete a substring from the current string. The substring to be deleted starts as position pos and is n characters in length. If n is larger than the number of characters between pos and the end of the string, it doesn’t do any harm. Thus, the default argument values cause deletion of “the rest of the string” if only a starting position is specified, and of the whole string if no arguments are specified. (The special value string::npos represents the maximum number of characters there can be in a string, and is thus one greater than the largest possible character position.)
string str13 = “abcdefghi”;
str12.erase (5,3);
cout << str12 << endl; // “abcdei”</td></tr>
<tr><td>replace</td><td>string& replace(size_type pos, size_type n, const string& str);
Delete a substring from the current string, and replace it with another string. The substring to be deleted is specified in the same way as in erase, except that there are no default values for pos and n.
string str14 = “abcdefghi”;
string str15 = “XYZ”;
str14.replace (4,2,str15);
cout << str14 << endl; // “abcdXYZghi”

Note: if you want to replace only a single character at a time, you should use the subscript operator ([]) instead.</td></tr>
<tr><td>find

Rfind</td><td>size_type find (const string& str, size_type pos=0) const;
size_type find (char ch, size_type pos=0) const;
size_type rfind (const string& str, size_type pos=npos) const;
size_type rfind (char ch, size_type pos=npos) const;
The find function searches for the first occurrence of the substring str (or the character ch) in the current string, starting at position pos. If found, return the position of the first character in the matching substring. If not found, return the value string::npos. The member function rfind does the same thing, but returns the position of the last occurrence of the specified string or character, searching backwards from pos.
string str16 = “abcdefghi”;
string str17 = “def”;
string::size_type pos = str16.find (str17,0);
cout << pos << endl; // 3
pos = str16.find (“AB”,0);
if (pos == string::npos) cout << “Not found” << endl;</td></tr>
</table>

find_first_of
 	
find_first_not_of
 	
find_last_of
 	
find_last_not_of
	size_type find_first_of (const string& str, size_type pos=0) const;
	size_type find_first_not_of (const string& str, size_type pos=0) const;
	size_type find_last_of (const string& str, size_type pos=npos) const;
	size_type find_last_not_of (const string& str, size_type pos=npos) const;
	Search for the first/last occurrence of a character that appears or does not appear in the **str** argument. If found, return the position of the character that satisfies the search condition; otherwise, return **string::npos**. The **pos** argument specifies the starting position for the search, which proceeds toward the end of the string (for "first" searches) or toward the beginning of the string (for "last" searches); note that the default values cause the whole string to be searched.
	string str20 = "Hello there";
	string str21 = "aeiou";
	pos = str20.find_first_of (str21, 0);
	cout << pos << endl; // 1
	pos = str20.find_last_not_of ("eHhlort");
	cout << pos << endl; // 5

substr	string substr (size_type pos, size_type n) const;
	Returns a substring of the current string, starting at position pos and of length n:
	string str18 = "abcdefghi"
	string str19 = str18.substr (6,2);
	cout << str19 << endl; // "gh"
	
	Note: if you want to retrieve only a single character at a time, you should use the subscript operator ([]) instead.

begin
 	
End
	iterator begin();
	const_iterator begin() const;
	iterator end();
	const_iterator end() const;
	Returns an iterator that specifies the position of the first character in the string (**begin**) or the position that is "just beyond" the last character in the string (**end**).
	
	Iterators are special objects that are used in many STL algorithms.

Non-member functions and algorithms

In addition to member functions of the **string** class, some non-member functions and STL algorithms can be used with strings; some common examples are:

getline	istream& getline (istream& is, string& str, char delim = ‘\n’);
	Reads characters from an input stream into a string, stopping when one of the following things happens:
	
	• An end-of-file condition occurs on the input stream
	• When the maximum number of characters that can fit into a string have been read
	• When a character read in from the string is equal to the specified delimiter (**newline** is the default delimiter); the delimiter character is removed from the input stream, but not appended to the string.
	
	The return value is a reference to the input stream. If the stream is tested as a logical value (as in an **if** or **while**), it is equivalent to **true** if the read was successful and **false** otherwise (e.g., end of file). [Note: some libraries do not implement **getline** correctly for use with keyboard input, requiring that an extra “carriage return” be entered to terminate the read. Some patches are available.]
	
	The most common use of this function is to do “line by line” reads from a file. Remember that the normal extraction operator (>>) stops on white space, not necessarily the end of an input line. The **getline** function can read lines of text with embedded spaces.
	
	vector<string> vec1;
	string line;
	vec1.clear();
	ifstream infile (“stl2in.txt”);
	while (getline(infile,line,’\n’))
	{
	  vec1.push_back (line);
	}

transform	OutputIterator transform (InputIterator first, InputIterator last, OutputIterator result, UnaryOperation op); Iterate through a container (here, a **string**) from **first** to just before **last**, applying the operation **op** to each container element and storing the results through the **result** iterator, which is incremented after each value is stored. This STL algorithm (from the **<algorithm>** library) is a little tricky, but simple uses are easy. For example, we can iterate through the characters in a string, modifying them in some way, and returning the modified characters back to their original positions. In this case, we set the **result** iterator to specify the beginning of the string. A common application is to convert a string to upper or lower case.

	string str22 = “This IS a MiXed CaSE stRINg”;
	transform (str22.begin(),str22.end(), str22.begin(), tolower);
	cout << “[” << str22 << “]” << endl; // [this is a mixed case string]
	
	Note that the **result** iterator must specify a destination that is large enough to accept all the modified values; here it is not a problem, since we’re putting them back in the same positions. The **tolower** function (along with **toupper**, **isdigit**, and other useful stuff) is in the **<cctype>** library; for more general case conversions, take a look at the **<locale>** library, which is beyond the scope of the present discussion.

Operators

A number of C++ operators also work with string objects:

=	The assignment operator may be used in several ways: • Assigning one string object’s value to another string object • string string_one = “Hello”; • string string_two; string_two = string_one; • Assigning a C++ string literal to a string object • string string_three; string_three = “Goodbye”; • Assigning a single character (char) to a string object • string string_four; • char ch = ‘A’; • string_four = ch; string_four = ‘Z’;
+	The “plus” operator concatenates: • two string objects • string str1 = “Hello “; • string str2 = “there”; string str3 = str1 + str2; // “Hello there” • a string object and a character string literal • string str1 = “Hello “; string str4 = str1 + “there”; • a string object and a single character

	• string str5 = "The End"; string str6 = str5 + '!';
+=	The "+=" operator combines the above assignment and concatenation operations in the way that you would expect, with a string object, a string literal, or a single character as the value on the right-hand side of the operator. string str1 = "Hello "; str1 += "there";
== != < > <= >=	The comparison operators return a **bool** (true/false) value indicating whether the specified relationship exists between the two operands. The operands may be: • two string objects • a string object and a character string literal
<<	The insertion operator writes the value of a string object to an output stream (e.g., **cout**). string str1 = "Hello there"; cout << str1 << endl;
>>	The extraction operator reads a character string from an input stream and assigns the value to a string object. string str1; cin >> str1;

[ ]
 	
(subscript)
	The subscript operator accesses one character in a string, for inspection or modification:
	
	string str10 = "abcdefghi";
	char ch = str10[3];
	cout << ch << endl; // 'd'
	str10[5] = 'X';
	cout << str10 << endl; // "abcdeXghi"

Question Bank

PART A (2 Marks)

Unit I

1. Define Object Oriented Programming (OOP).

Object Oriented Programming is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.

2. List out the basic concepts of Object Oriented Programming.

- Objects
- Classes
- Data Abstraction and Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

3. Define Objects.

Objects are the basic run time entities in an object oriented system. They are instance of a class. They may represent a person, a place etc that a program has to handle. They may also represent user-defined data. They contain both data and code.

4. Define Class.

Class is a collection of objects of similar data types. Class is a user-defined data type. The entire set of data and code of an object can be made a user defined type through a class. The general form of a class is,

```
class class_name
{
private:
variable declarations;
function declaration;
public:
variable declarations;
function declaration;
};
```

5. Define Encapsulation and Data Hiding.

The wrapping up of data and functions into a single unit is known as data encapsulation. Here the data is not accessible to the outside world. The insulation of data from direct access by the program is called data hiding or information hiding.

6. Define Data Abstraction.

Abstraction refers to the act of representing the essential features without including the background details or explanations.

7. Define data members and member functions.

The attributes in the objects are known as data members because they hold the information. The functions that operate on these data are known as methods or member functions.

8. State Inheritance.

Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification and provides the idea of reusability. The class which is inherited is known as the base or super class and class which is newly derived is known as the derived or sub class.

9. State Polymorphism.

Polymorphism is an important concept of OOPs. Polymorphism means one name, multiple forms. It is the ability of a function or operator to take more than one form at different instances.

10. List and define the two types of Polymorphism.

- ***Operator Overloading*** – The process of making an operator to exhibit different behaviors at different instances.
- ***Function Overloading*** – Using a single function name to perform different types of tasks. The same function name can be used to handle different number and different types of arguments.

11. State Dynamic Binding.

Binding refers to the linking of procedure call to the code to be executed in response to the call. Dynamic Binding or Late Binding means that the code associated with a given procedure call is known only at the run-time.

12. Define Message Passing.

Objects communicate between each other by sending and receiving information known as messages. A message to an object is a request for execution of a procedure. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

13. List out some of the benefits of OOP.

- Eliminate redundant code
- Saves development time and leads to higher productivity
- Helps to build secure programs
- Easy to partition work
- Small programs can be easily upgraded to large programs
- Software complexity can easily be managed

14. List out the applications of OOP.

- Real time systems
- Simulation and modeling

- Object oriented databases
- Hypertext, Hypermedia and expertext
- AI and expert systems
- Neural networks and parallel programming
- CIM/CAM/CAD systems

15. Define C++.

C++ is an object oriented programming language developed by Bjarne Stroustrup. It is a super set of C. Initially it was known as “C with Classes”. It is a versatile language for handling large programs.

16. What are the input and output operators used in C++?

The identifier cin is used for input operation. The input operator used is >>, which is known as the extraction or get from operator. The syntax is,

cin >> n1;

The identifier cout is used for output operation. The input operator used is <<, which is known as the insertion or put to operator. The syntax is,

cout << “C++ is better than C”;

17. List out the four basic sections in a typical C++ program.

Include files
Class declaration
Member functions definition
Main function program

18. State the use of void in C++.

The two normal uses of void are

i) To specify the return type of the function when it is not returning a value

ii) To indicate an empty argument list to a function

19. List out the new operators introduced in C++.

- :: Scope resolution operator
- ::* Pointer to member declarator
- ->* Pointer to member operator
- .* Pointer to member operator
- delete Memory release operator
- endl Line feed operator
- new Memory allocation operator
- setw Field width operator

20. What is the use of scope resolution operator?

A variable declared in an inner block cannot be accessed outside the block. To resolve this problem the scope resolution operator is used. It can be used to uncover a

hidden variable. This operator allows access to the global version of the variable. It takes the form, :: variable-name

21. List out the memory differencing operator.

- ::* To declare a pointer to the member of the class
- ->* To access a member using object name and a pointer to that member
- .* To access a member using a pointer to the object and a pointer to that member

22 List the access modes used within a class.

i) ***Private*** – The class members are private by default. The members declared private are completely hidden from the outside world. They can be accessed from only within the class.
ii) ***Public*** – The class members declared public can be accessed from any where.
iii) ***Protected*** – The class members declared protected can be access from within the class and also by the friend classes.

23 How can we access the class members?

The class members can be accessed only when an object is created to that class. They are accessed with the help of the object name and a dot operator. They can be accessed using the general format,

Object_name.function_name (actual_arguments);

24 Where can we define member functions?

Member functions can be defined in two places:
i) ***Outside the class definition*** – The member functions can be defined outside the class definition with the help of the scope resolution operator.
The general format is given as,

```
return_type class_name :: function_name (argument declaration)
{
        function body
}
```

ii) ***Inside the class definition*** – The member function is written inside the class in place of the member declaration. They are treated as inline functions.

25. What are the characteristics of member functions?

The various characteristics of member functions are,

- Different classes can use the same function name and their scope can be resolved using the membership label.
- Member functions can access the private data of a class while a nonmember function cannot.
- A member function can call another member function directly without using a dot operator.

26. What are inline functions?

An inline function is a function that is expanded in line when it is invoked. Here, the compiler replaces the function call with the corresponding function code. The inline function is defined as,

```
inline function-header
{
        function body
}
```

27. Why do we use default arguments?

The function assigns a default value to the parameter which does not have a matching argument in the function call. They are useful in situations where some arguments always have the same value.
e.g., float amt (float P, float n, float r = 0.15);

28. State the advantages of default arguments.

The advantages of default arguments are,

- We can use default arguments to add new parameters to the existing function.
- Default arguments can be used to combine similar functions into one.

29. Define function overloading.

A single function name can be used to perform different types of tasks. The same function name can be used to handle different number and different types of arguments. This is known as function overloading or function polymorphism.

30. List out the limitations of function overloading.

We should not overload unrelated functions and should reserve function overloading for functions that perform closely related operations.

31. Define friend function?

A function that has access to the private members of a class but is not itself a member of the class. An entire class can be a friend of another class.

32. Define friend function?

An outside function can be made a friend to a class using the qualifier ‘friend’. The function declaration should be preceded by the keyword friend. A friend function has full access rights to the private members of a class.

33. List out the special characteristics of a friend function.

- It is not in the scope of a class in which it is declared as friend.
- It cannot be called using the object of that class.
- It can be invoked without an object.
- It cannot access the member names directly and uses the dot operator.
- It can be declared as either public or private.

- It has the objects as arguments.

34. What is constant member function?

If a member function does not alter any data in the class, then that may declare as ***const*** member function.

```
return-type function-name( ) const;
```

35. What are constant arguments?

The qualifier const tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated.

36. What are the properties of a static data member?

The properties of a static data member are,

- It is initialized to zero when the first object is created and no other initialization is permitted.
- Only one copy of that member is created and shared by all the objects of that class.
- It is visible only within the class, but the life time is the entire program.

37. What are the properties of a static member function?

- A static member function can access only other static members declared in the same class.
- It can be called using the class name instead of objects as follows,
  class_name :: function_name;

38. Define constant object.

A constant object is created using ***const*** keyword before object declaration. A constant object can call only constant member function.

```
const classname object-name;
```

39. What is nesting of classes?

A class can contain objects of other classes as its members. It is called nesting of classes.

```
Class A{….};
Class B
{
        A a;
};
```

40. Define local classes.

Classes can be defined and used inside a function or a block. Such classes are called local classes.

```
void test( int a)        //function
```

```
{
  …..
  class student          //local class
  {
        ….
        …
  };
  …
  student s1(a);         //create local class object
  …..
}
```

Unit II

1. Define Constructor.

A constructor is a special member function whose task is to initialize the objects of its class. It has the same name as the class. It gets invoked whenever an object is created to that class. It is called so since it constructs the values of data members of the class.

2. List some of the special characteristics of constructor.

- Constructors should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types
- They cannot be inherited.

3. Give the various types of constructors.

There are four types of constructors. They are

- Default constructors – A constructor that accepts no parameters
- Parameterized constructors – The constructors that can take arguments
- Copy constructor – It takes a reference to an object of the same class as itself as an argument
- Dynamic constructors – Used to allocate memory while creating objects

4. What are the ways in which a constructor can be called?

The constructor can be called by two ways. They are,

- By calling the constructor explicitly
  e.g., integer int1 = integer (0, 100);
- By calling the constructor implicitly
  e.g., integer int1 (0, 100);

5. What are the kinds of constructors that we call?

- Constructor without arguments
- Constructor with arguments

6. Define dynamic constructor?

Allocation of memory to objects at the time of their construction is known as dynamic constructor.

7. What is the advantage of using dynamic initialization?

The advantage of using dynamic initialization is that various initialization formats can be provided using overloaded constructor.

8. State dynamic initialization of objects.

Class objects can be initialized dynamically. The initial values of an object may be provided during run time. The advantage of dynamic initialization is that various initialization formats can be used. It provides flexibility of using different data formats.

9. Define Destructor.

A destructor is used to destroy the objects that have been created by a constructor. It is a special member function whose name is same as the class and is preceded by a tilde '~' symbol.

10. List the difference between constructor and destructor?

Constructor can have parameters. There can be more than one constructor. Constructors is invoked when from object is declared. Destructor has no parameters. Only one destructor is used in class. Destructor is invoked up on exit program.

11. Give the general form of an operator function.

The general form of an operator function is given as,

```
return-type class-name :: operator op (arg-list)
{
        function body
}
```

Where,

Return-type -> type of value returned
operator -> keyword
op -> operator being overloaded

12. List some of the rules for operator overloading.

- Only existing operators can be overloaded.
- We cannot change the basic meaning of an operator.
- The overloaded operator must have at least one operand.
- Overloaded operators follow the syntax rules of the original operators.

13. Give the operator in C++ which cannot be overloaded?

- Sizeof ->size of operator
- :: ->scope resolution operator
- ?: -> conditional operator
- . ->Membership operator
- .* ->pointer to member operator

14. What are the steps that involves in the process of overloading?

- Creates a class that defines the data type that is to be used in the overloading operation.
- Declare the operator function operator op ( ) in the public part of a class.
- Define the operator function to implement the required operation.

15. What are the restriction and limitations overloading operators?

Operator function must be member functions are friend functions. The overloading operator must have at least one operand that is of user defined data type.

16. Define unary and binary operator overloading?

Overloading without explicit arguments t an operator function is known as Unary operator overloading and overloading with a single explicit argument is known as binary operator overloading.

17. Explain overloading of new and delete operators?

The memory allocation operators new and delete can be overloaded to handle memory resource in a customized way. The main reason for overloading these functions is to increase the efficiency of memory management.

18. Give the syntax for overloading with friend functions?

```
Friend return type operator op(arguments)
{
        Body of the function
}
```

19. Define type conversion?

A conversion of value from one data type to another type is known as type conversion.

20. What are the types of type conversions?

There are three types of conversions. They are

- Conversion from basic type to class type – done using constructor
- Conversion from class type to basic type – done using a casting operator
- Conversion from one class type to another – done using constructor or casting operator

21. What are the conditions should a casting operator satisfy?

The conditions that a casting operator should satisfy are,

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

22. When and how the conversion function exists?

To convert the data from a basic type to user defined type. The conversion should be defined in user defined object's class in the form of a constructor. The constructor function takes a single argument of basic data type.

Unit III

1. List the application of templates.

It is a new concept which enables to define generic classes and functions and thus provides support for generic programming.

2. Name the types of templates.

i) Class template
ii) Function template

3. Define class template. Write the syntax.

Process of creating a generic class using template with an anonymous type is called as class template.

```
template<class T>
class classname
{
// class member specification with anonymous type T
}
```

4. What is generic programming?

Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.

5. What is template class or instantiation?

A specific class created from class template is known as template class. The process of creating template class is known as instantiation.

6. Define class template with multiple parameters.

A class template use more than one generic data type is class template with multiple parameters.

```
template<class T1, class T2, ..>
class classname
{
// class member specification with anonymous type T
}
```

7. What is function template? Give syntax.

Process of creating a generic function using template with an anonymous type is called as function template.

```
template<class T>
return-type function-name(arg of type T)
```

```
{
// body of function with anonymous type T
}
```

8. What is function template?

A specific function created from function template is known as template function.

9. Define function template with multiple parameters.

A function template use more than one generic data type is function template with multiple parameters.

```
template<class T1, class T2, ..>
return-type function-name(arg of type T)
{
// body of function with anonymous type T
}
```

10. Define non-type template arguments.

In addition to the type argument T in template, we can also use other arguments such as strings, function names, constant expressions and built-in-types. This is known as non-type template arguments.

```
template<class T, int variable>
class classname
{
        ….
};
```

11. List the types of error.

i) Logical error
ii) Syntax error

12. List some examples for exceptions.

i) Divide by zero error.
ii) Accessing an array outside its memory bounds.
iii) Running out-of memory or disk space.

13. How the exception can be handled?

i) Find the problem (hit the exception)
ii) Inform that an error has occurred (throw the exception)
iii) Receive the error information (catch the exception)
iv) Take corrective actions (handle the exception)

14. List the types of exceptions and define them.

i) Synchronous Exception – exception that belongs to “out-of-range” and “over flow”.
ii) Asynchronous Exception - exception that belongs to keyboard interrupts.

15. Write the general form of try-catch-throw paradigms.

```
try
{
  throw exception;
}
catch(type arg)
{
  // catch body
}
```

16. List the different throwing mechanism.

- throw(exception);
- throw exception;
- throw;
-

17. List different catch mechanism.

i) Multiple catch – A catch block which can catch the matching type exception.

```
try{ ……}
catch(type1 arg)
{ …….}
.
.
catch(typeN arg)
{ ………}
```

ii) Catch all - A catch block which can caught all exception.

```
catch(…)
{
….
…
}
```

18. What is re-throwing an exception?

An exception can be re-thrown using a statement called throw. The re-thrown exception will be caught only by the next try-catch block sequence and not by the same try-catch sequence.

19. Write the syntax to test throwing restrictions.

Some functions may be restricted in throwing some types of exceptions. That can be done simply by adding a throw list clause to the function definition.

Syntax:

```
Type function_name(arg-list) throw(type-list)
{
……
}
```

Unit IV

1. Define inheritance?

The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class and the new one is called the derived class or the subclass.

2. Give the syntax for inheritance.

The syntax of deriving a new class from an already existing class is given by,

```
class derived-class : visibility-mode base-class
{
body of derived class
}
```

3. What are the types in inheritance?

i. Single inheritance
ii. Multiple inheritance
iii. Multilevel inheritance
iv. Hierarchical inheritance
v. Hybrid inheritance

4. Explain single inheritance?

A derived class with only one base class is called single inheritance.

5. What is multiple inheritance?

A derived class with more than one base class is called multiple inheritance.

6. Define hierarchical inheritance?

One class may be inherited by more than one class. This process is known as hierarchical inheritance.

7. What is hybrid inheritance?

There could be situations where we need to apply two or more type of inheritance to design a program. This is called hybrid inheritance.

8. What is multilevel inheritance?

The mechanism of deriving a class from another derived class is known as multilevel inheritance.

9. What is virtual base class?

The child has two direct base classes parent1 and parent2 which themselves have common base class grand parent. The child inherits the traits of grand parent via two separate paths. It can also inherit directly shown by the broken line. The grand parent is sometimes referred to as indirect base class.

10. What is abstract class?

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class. It is a design concept is program development and provides a base upon which other classes may be built.

11. What are the types of polymorphism?

The two types of polymorphism are,
i) Compile time polymorphism – The compiler selects the appropriate function for a particular call at the compile time itself. It can be achieved by function overloading and operator overloading.
ii) Run time Polymorphism - The compiler selects the appropriate function for a particular call at the run time only. It can be achieved using virtual functions.

12. Define 'this' pointer.
A 'this' pointer refers to an object that currently invokes a member function. For e.g., the function call a. show () will set the pointer 'this' to the address of the object 'a'.

13. What is a virtual function?
When a function is declared as virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer.

14. What are the rules for virtual function?
1. They cannot be static members
2. They are access by using object pointers
3. A virtual function can be a friend of another class.

15. What is a pure virtual function?
A virtual function, equated to zero is called a pure virtual function. It is a function declared in a base class that has no definition relative to the base class.

16. How can a private member be made inheritable?
A private member can be made inheritable by declaring the data members as protected. When declared as protected the data members can be inherited by the friend classes.

17. What is polymorphism? What is the difference between compile time and runtime polymorphism?
The ability of an object to behave differently in different context given in the same message is known as polymorphism.
Polymorphism achieved using operator overloading and function overloading is known as compile time polymorphism.
Polymorphism achieved using virtual function is known as runtime polymorphism.

18. Define dynamic binding.

Linking the function at runtime is known as dynamic binding. Virtual functions are dynamically bound.

19. Define static binding.

Linking a function during linking phase is known as static binding. Normal functions are statically bound.

20. What is this pointer?

A unique keyword called this to represent an object has invokes a member function.

21. Differentiate between virtual functions and pure virtual functions?

A function defined with virtual keyword in the base class is known as ***virtual function***.

A virtual function with '=0' in place of body in the class is known as ***pure virtual function***. They may not have a body. It is possible to have a body of pure virtual function defined outside the class.

22. What are the rules for virtual function?

1. They cannot be static members
2. They are access by using object pointers
3. A virtual function can be a friend of another class.

23. What is virtual table?

A table consisting of pointers of all virtual functions of the class is called as virtual table. Every class with at least one virtual function will have one copy of the virtual table.

24. Why do we need RTTI?

Run Time Type Information is a mechanism to decide the type of the object at runtime.

25. What are polymorphic object?

If we have a class containing a virtual function, it is possible to point to a derived class object using the base class pointer and manipulate the object. Such manipulatable objects are known as polymorphic objects.

26. What is the need for type_info object? What is the role of typeid operator in the RTTI?

Type_info is an object associated with every built-in and user defined type, and also with polymorphic type. This object describes the type of the object. This object type for polymorphic type is only available when RTTI is enabled.

Typeid operator can be applied to any object or class or built-in data type. It returns the type_info object associated with the object under consideration.

27. List the attributes of type_info object.

i) Name() function — This function returns the type name in a string.

ii) Operator = = — This can compare the types of two different type_info objects.

iii) Operator! = — This pointer compares two typeid objects and returns true if they are of different type.

28. Write the syntax of typeid.

typeid (object);

29. Give the syntax of dynamic casting.

dynamic_cast< ToObjectPtr or Ref> (FromObjectPtr or Ref)

30. Is it possible to convert a problem solved using dynamic_cast to a program using typeid? Give your views on such conversion.

Yes, it is possible to convert a problem solved using dynamic_cast to a program using typeid. This is done by two steps:

Step 1: Check the types of two arguments using typeid and operator = =.

Step 2: If answer is yes, use a plain vanilla c type casting to cast.

31. Define cross casing.

Cross casting refers to casting from derived class to proper base class when there are multiple base classes in the case of multiple-inheritance.

When a multiple derived class object is pointed to by one of its base class pointers, casting from one base class pointer into another base class pointer is known as cross casting.

32. What is down casing?

Casing from base class pointer to derived class pointer is known as down-casting.

33. What is dynamic_cast?

This is a new casting operator provided in C++ which casts a polymorphic object into another if it is correct to do it.

Unit V

1. What is a Stream?

A stream is an object where a program can either insert or extract characters to or from it. The standard input and output stream objects of C++ are declared in the header file *iostream*.

2. What is Output Stream

ios_base → ios → **ostream** → iostream

→ **ofstream**
→ **ostringstream**

`ostream` objects are stream objects used to write and format output as sequences of characters

3. *What is a Manipulator?*
Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer's choice of display.
11. Define Namespaces
Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.
The format of namespaces is:

```
namespace identifier
{
entities
}
```

Where `identifier` is any valid identifier and `entities` is the set of classes, objects and functions that are included within the namespace. For example:
12. Define Standard Template Library
The Standard Template Libraries (STL's) are a set of C++ template classes to provide common programming data structures and functions such as doubly linked lists (list), paired arrays (map), expandable arrays (vector), large string storage and manipulation (rope), etc.

PART B - (16 Marks)

Unit I

1. i) Explain with the Basic Concepts of object oriented programming. (8 Marks)
   ii) Explain the use of constant pointers and pointers to constant with an example. (8)
2. i) Difference between class and struct and also illustrate with an example. (8 Marks)
   ii) What are the difference between pointers to constants and constant to pointers? (8)
3. i) Write a C++ program using inline function. (8 Marks)
   ii)Write a C++ program to illustrate the static function. (8 Marks)
4. i) Explain briefly about function overloading with a suitable example. (8 Marks)
   ii)Discuss constant and volatile functions. (8 Marks)
5. i) Explain Nested classes and Local classes with an example. (8 Marks)
   ii)Explain about call-by-reference and return by reference with program. (8 Marks)

Unit II

1. Explain about Constructor and Destructor with suitable C++ coding. (16 Marks)
2. Explain about Copy Constructor with an example. (16 Marks)
3. Explain Explicit Constructors, Parameterized Constructors and Multiple constructors with suitable examples. (16 Marks)
4. How to achieve operator overloading through friend function? (16 Marks)

5. Write a program using friends function for overloading << and >> operators. (16)
6. Explain type conversion with examples. (16 Marks)
7. Write a program for overloading the assignment operators. (16)

Unit III

1. Explain the Function template. (16 Marks)
2. Explain the Class template. (16 Marks)
3. i) What is the need for Exceptional Handling (8 Marks)
   ii) What are the components of Exception Handling Mechanism with suitable example
4. Explain the following function
   i. rethrow ( ) (4 Marks)
   ii. terminate ( ) (4 Marks)
   iii. unexpected ( ) and uncaught_exception () (8 Marks)
5. i)What are the disadvantages of the Exception handling mechanism? (8 Marks)
   ii)What are specifications in throw? In which case are they needed? (8 Marks)
6.i) When do we need multiple catch blocks for a single try block? Give an example (16)

Unit IV

1. i) Explain the different types of polymorphism. (10 Marks)
   ii) How to use RTTI for template objects? (6 Marks)
2. Explain various types of inheritance. (16 Marks)
3. Describe pure virtual function with an example. (16 Marks)
4. i) Write a C++ program using this pointer. (8 Marks)
   ii) Write a C++ program using Dynamic casting. (8 Marks)
5. Explain briefly about
   a. Cross casting (8 Marks)
   b. Down casting (8 Marks)

Unit V

1. i) What are streams? Why they are useful? (8 Marks)
   ii) What is the importance of ios member flags in formatting IO? (8 Marks)
2. Explain about Formatted and Unformatted IO with suitable Example (16 Marks)
3. What is the manipulator? Differences between Manipulators and ios Functions. (16)
4. Explain the process of open, read, write, and close files? (16 Marks)
5. Explain the roles of seekg (), seekp (), tellg (), tellp () functions in the process of random access in a binary file. (16 Marks)
6. i) How can we determine errors while dealing with files? (8 Marks)
   ii) How can we open a text file and read from it? (4 Marks)
   iii) How can we open a binary file and write to it? (4 Marks)
7. Explain about the STD Namespaces. (16Marks)
9. Explain the different constructions of string objects (16 Marks)
10. Explain the standard Template Library and how it is working? (16 Marks)