INSTRUCTION MANUAL

Object Oriented Programming Using C++ Lab

(BTCS-309)

Prepared by

Er. Gagandeep Kaur
Assistant Professor (CSE)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

GURU NANAK DEV ENGINEERING COLLEGE
LUDHIANA – 141006

DECLARATION

This Manual of Object Oriented Programming Using C++ Lab (BTCS-309) has been prepared by me as per syllabus of Object Oriented Programming Using C++ Lab (BTCS-309).

Signature

Syllabus

1. [Classes and Objects] Write a program that uses a class where the member functions are defined inside a class.

2. [Classes and Objects] Write a program that uses a class where the member functions are defined outside a class.

3. [Classes and Objects] Write a program to demonstrate the use of static data members.

4. [Classes and Objects] Write a program to demonstrate the use of const data members.

5. [Constructors and Destructors] Write a program to demonstrate the use of zero argument and parameterized constructors.

6. [Constructors and Destructors] Write a program to demonstrate the use of dynamic constructor.

7. [Constructors and Destructors] Write a program to demonstrate the use of explicit constructor.

8. [Initializer Lists] Write a program to demonstrate the use of initializer list.

9. [Operator Overloading] Write a program to demonstrate the overloading of increment and decrement operators.

10. [Operator Overloading] Write a program to demonstrate the overloading of binary arithmetic operators.

11. [Operator Overloading] Write a program to demonstrate the overloading of memory management operators.

12. [Typecasting] Write a program to demonstrate the typecasting of basic type to class type.

13. [Typecasting] Write a program to demonstrate the typecasting of class type to basic type.

14. [Typecasting] Write a program to demonstrate the typecasting of class type to class type.

15. [Inheritance] Write a program to demonstrate the multilevel inheritance.

16. [Inheritance] Write a program to demonstrate the multiple inheritance.

17. [Inheritance] Write a program to demonstrate the virtual derivation of a class.

18. [Polymorphism] Write a program to demonstrate the runtime polymorphism.

19. [Exception Handling] Write a program to demonstrate the exception handling.

20. [Templates and Generic Programming] Write a program to demonstrate the use of function template.

21. [Templates and Generic Programming] Write a program to demonstrate the use of class template.

22. [File Handling] Write a program to copy the contents of a file to another file byte by byte.
The name of the source file and destination file should be taken as command-line arguments,

23. [File Handling] Write a program to demonstrate the reading and writing of mixed type of data.

24. [File Handling] Write a program to demonstrate the reading and writing of objects.

INDEX

S. NO.	PRACTICAL	Page Number
1	**[Classes and Objects]** Write a program that uses a class where the member functions are defined inside a class.	1
2	**.[Classes and Objects]** Write a program that uses a class where the member functions are defined outside a class.	
3	**[Classes and Objects]** Write a program to demonstrate the use of static data members.	
4	**[Classes and Objects]** Write a program to demonstrate the use of const data members.	
5	**[Constructors and Destructors]** Write a program to demonstrate the use of zero argument and parameterized constructors.	8
6	**[Constructors and Destructors]** Write a program to demonstrate the use of dynamic constructor.	
7	**[Constructors and Destructors]** Write a program to demonstrate the use of explicit constructor.	
8	**[Initializer Lists]** Write a program to demonstrate the use of initializer list.	
9	**[Operator Overloading]** Write a program to demonstrate the overloading of increment and decrement operators.	16
10	**[Operator Overloading]** Write a program to demonstrate the overloading of binary arithmetic operators.	
11	**[Operator Overloading]** Write a program to demonstrate the overloading of memory management operators.	
12	**[Typecasting]** Write a program to demonstrate the typecasting of basic type to class type.	23
13	Write a program to demonstrate the typecasting of class type to basic type.	
14	**[Typecasting]** Write a program to demonstrate the typecasting of class type to class type.	
15	**[Inheritance]** Write a program to demonstrate the multilevel inheritance.	27
16	**[Inheritance]** Write a program to demonstrate the multiple inheritance.	
17	**[Inheritance]** Write a program to demonstrate the virtual derivation of a class.	
18	**[Polymorphism]** Write a program to demonstrate the runtime polymorphism.	
19	**[Exception Handling]** Write a program to demonstrate the exception handling.	35
20	**[Templates and Generic Programming]** Write a program to demonstrate the use of function template.	40
21	**[Templates and Generic Programming]** Write a program to demonstrate the use of class template.	
22	**[File Handling]** Write a program to copy the contents of a file o another file byte by byte. The name of the source file and destination file should be taken as command-line arguments,	44
23	**[File Handling]** Write a program to demonstrate the reading and writing of mixed type of data.	
24	**[File Handling]** Write a program to demonstrate the reading and writing of objects.	

Classes and Objects:-

1. [Classes and Objects] Write a program that uses a class where the member functions are defined inside a class.

2. [Classes and Objects] Write a program that uses a class where the member functions are defined outside a class.

3. [Classes and Objects] Write a program to demonstrate the use of static data members.

4. [Classes and Objects] Write a program to demonstrate the use of const data members.

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

C++ Class Definitions:

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows:

```
class Box
{
  public:
    double length;   // Length of a box double breadth;  // Breadth of a box
    double height;   // Height of a box
};
```

The keyword **public** determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub- section.

Define C++ Objects:

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

```
Box Box1;        // Declare Box1 of type Box
Box Box2;        // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members:

The public data members of objects of a class can be accessed using the direct member access operator (.).

Let us try the following example to make the things clear:

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

class Box
{
public:
   double length;   // Length of a box double breadth;  // Breadth of a box double height;   // Height
   of a box
};

int main( )
{
  Box Box1;        // Declare Box1 of type Box
  Box Box2;        // Declare Box2 of type Box
  double volume = 0.0;     // Store the volume of a box here

  // box 1 specification Box1.height = 5.0; Box1.length = 6.0;
  Box1.breadth = 7.0;

  // box 2 specification Box2.height = 10.0; Box2.length = 12.0; Box2.breadth = 13.0;
  // volume of box 1
  volume = Box1.height * Box1.length * Box1.breadth;
  cout << "Volume of Box1 : " << volume <<endl;

  // volume of box 2
  volume = Box2.height * Box2.length * Box2.breadth;
```

```
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

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

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

It is important to note that private and protected members can not be accessed directly using direct member access operator (.).

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Let us take previously defined class to access the members of the class using a member function instead of directly accessing them:

```
class Box
{
   public:
      double length;         // Length of a box double breadth;        // Breadth of a box double height;
      // Height of a box
      double getVolume(void);// Returns box volume
};
```

Member functions can be defined within the class definition or separately using **scope resolution operator, ::**. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier. So either you can define **Volume()** function as below:

```
class Box
{
   public:
      double length;      // Length of a box double breadth;     // Breadth of a box
      double height;      // Height of a box

      double getVolume(void)
      {
         return length * breadth * height;
      }
};
```

If you like you can define same function outside the class using **scope resolution operator, ::** as follows:

```
double Box::getVolume(void)
 {
    return length * breadth * height;
 }
```

Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (**.**) on a object where it will manipulate data related to that object only as follows:

```
    Box myBox;          // Create an object

    myBox.getVolume();  // Call member function for the object
```

Let us put above concepts to set and get the value of different class members in a class:

```
    #include <iostream>

    using namespace std;

    class Box
    {
      public:
        double length;        // Length of a box double breadth;        // Breadth of a box
        double height;        // Height of a box

        // Member functions declaration double getVolume(void);
        void setLength( double len ); void setBreadth( double bre ); void setHeight( double hei );
    };

    // Member functions definitions double Box::getVolume(void)
    {
       return length * breadth * height;
    }void Box::setLength( double len )
    {
       length = len;
    }

    void Box::setBreadth( double bre )
    {
       breadth = bre;
```

```
}

void Box::setHeight( double hei )
{
   height = hei;
}

// Main function for the program int main( )
{
  Box Box1;              // Declare Box1 of type Box
  Box Box2;              // Declare Box2 of type Box
  double volume = 0.0;     // Store the volume of a box here

  // box 1 specification Box1.setLength(6.0); Box1.setBreadth(7.0);
  Box1.setHeight(5.0);

  // box 2 specification Box2.setLength(12.0); Box2.setBreadth(13.0);
  Box2.setHeight(10.0);

  // volume of box 1
  volume = Box1.getVolume();
  cout << "Volume of Box1 : " << volume <<endl;

  // volume of box 2
  volume = Box2.getVolume();cout << "Volume of Box2 : " << volume <<endl;
      return 0;
   }
```

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

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

The const Keyword:

You can use **const** prefix to declare constants with a specific type as follows:

```
const type variable = value;
```

Following example explains it in detail:

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

```
int main()
{
   const int  LENGTH = 10;
   const int  WIDTH  = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   cout << area;
   cout << NEWLINE;
   return 0;
}
```

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

```
50
```

Constructors and Destructors and Initializer List:-

5. **[Constructors and Destructors]** Write a program to demonstrate the use of zero argument and parameterized constructors.
6. **[Constructors and Destructors]** Write a program to demonstrate the use of dynamic constructor.
7. **[Constructors and Destructors]** Write a program to demonstrate the use of explicit constructor.
8. **[Initializer Lists]** Write a program to demonstrate the use of initializer list.

The Class Constructor:

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Following example explains the concept of constructor:

```
#include <iostream>
using namespace std;
class Line
{
  public:
    void setLength( double len );
    double getLength( void );
    Line();  // This is the constructor

  private:
    double length;
};
// Member functions definitions including constructor
Line::Line(void)
{
   cout << "Object is being created" << endl;
}
void Line::setLength( double len )
{
   length = len;
}
double Line::getLength( void )
{
   return length;
```

```
}
// Main function for the program int main( )
{
   Line line;

   // set line length line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

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

```
Object is being created
Length of line : 6
```

Parameterized Constructor:

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example:

```
#include <iostream>

using namespace std;

class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line(double len);  // This is the constructor

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line( double len)
{
```

```
   cout << "Object is being created, length = " << len << endl;
   length = len;
}

void Line::setLength( double len )
{
   length = len;
}

double Line::getLength( void )
{
   return length;
}
// Main function for the program int main( )
{
  Line line(10.0);

  // get initially set length.
  cout << "Length of line : " << line.getLength() <<endl;
  // set line length again line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;

  return 0;
}
```

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

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one.If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here:

```
classname (const classname &obj) {
// body of constructor
}
```

Here, **obj** is a reference to an object that is being used to initialize another object.

```
#include <iostream>

using namespace std;

class Line
{
  public:
    int getLength( void );
    Line( int len );            // simple constructor
    Line( const Line &obj);  // copy constructor
    ~Line();                    // destructor

  private:
    int *ptr;
};

// Member functions definitions including constructor
Line::Line(int len)
{
   cout << "Normal constructor allocating ptr" << endl;
   // allocate memory for the pointer;
   ptr = new int;
   *ptr = len;
}

Line::Line(const Line &obj)
{
   cout << "Copy constructor allocating ptr." << endl;
   ptr = new int;
  *ptr = *obj.ptr; // copy the value
}
```

```
Line::~Line(void)
{
    cout << "Freeing memory!" << endl;
    delete ptr;
}
int Line::getLength( void )
{
    return *ptr;
}

void display(Line obj)
{
   cout << "Length of line : " << obj.getLength() <<endl;
}

// Main function for the program int main( )
{
   Line line(10);

   display(line);

   return 0;
}
```

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

Normal constructor allocating ptr Copy constructor allocating ptr. Length of line : 10
Freeing memory! Freeing memory!

Let us see the same example but with a small change to create another object using existing object of the same type:

```
#include <iostream>

using namespace std;

class Line
```

```
{
  public:
    int getLength( void );
    Line( int len );             // simple constructor
    Line( const Line &obj);  // copy constructor
    ~Line();                     // destructor

  private:
    int *ptr;
};

// Member functions definitions including constructor
Line::Line(int len)
{
   cout << "Normal constructor allocating ptr" << endl;
   // allocate memory for the pointer;
   ptr = new int;
   *ptr = len;
}

Line::Line(const Line &obj)
{
   cout << "Copy constructor allocating ptr." << endl;
   ptr = new int;
  *ptr = *obj.ptr; // copy the value
}

Line::~Line(void)
{
   cout << "Freeing memory!" << endl;
   delete ptr;
}
int Line::getLength( void )
{
   return *ptr;
}

void display(Line obj)
```

```
{
  cout << "Length of line : " << obj.getLength() <<endl;
}

// Main function for the program int main( )
{
  Line line1(10);

  Line line2 = line1; // This also calls copy constructor

  display(line1);
  display(line2);

  return 0;
}
```

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

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Copy constructor allocating ptr. Length of line : 10
Freeing memory!
Copy constructor allocating ptr. Length of line : 10
Freeing memory! Freeing memory!
Freeing memory!
```

Using Initialization Lists to Initialize Fields:

In case of parameterized constructor, you can use following syntax to initialize the fields:

```
Line::Line( double len): length(len)
{
   cout << "Object is being created, length = " << len << endl;
}
```

Above syntax is equal to the following syntax:

```
Line::Line( double len)
{cout << "Object is being created, length = " << len << endl;
   length = len;
}
```

If for a class C, you have multiple fields X, Y, Z, etc., to be initialized, then use can use same

syntax and separate the fields by comma as follows:

```
C::C( double a, double b, double c): X(a), Y(b), Z(c)
{
  ....
}
```

The Class Destructor:

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explains the concept of destructor:

```
#include <iostream>

using namespace std;

class Line
{
  public:
    void setLength( double len );
    double getLength( void );
    Line();   // This is the constructor declaration
    ~Line();  // This is the destructor: declaration

  private:
    double length;
};

// Member functions definitions including constructor
Line::Line(void)
{
   cout << "Object is being created" << endl;
}Line::~Line(void)
{
  cout << "Object is being deleted" << endl;
}
```

```
void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}
// Main function for the program int main( )
{
   Line line;

   // set line length line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

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

```
Object is being created
Length of line : 6
Object is being deleted
```

Operator Overloading:-

9. [Operator Overloading] Write a program to demonstrate the overloading of increment and decrement operators.

10. [Operator Overloading] Write a program to demonstrate the overloading of binary arithmetic operators.

11. [Operator Overloading] Write a program to demonstrate the overloading of memory management operators.

Operators overloading in C++:

You can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

```
Box operator+(const Box&);
```

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows:

```
Box operator+(const Box&, const Box&);
```

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below:

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

class Box
{
  public:

    double getVolume(void)
    {
      return length * breadth * height;
    }
    void setLength( double len )
    {
       length = len;
    }



void setBreadth( double bre )
{
```

```
        breadth = bre;
      }
      void setHeight( double hei )
      {
        height = hei;}
      // Overload + operator to add two Box objects. Box operator+(const Box& b)
      {
        Box box;
        box.length = this->length + b.length;
        box.breadth = this->breadth + b.breadth;
        box.height = this->height + b.height;
        return box;
      }
    private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
  };
  // Main function for the program int main( )
  {
    Box Box1;               // Declare Box1 of type Box
    Box Box2;               // Declare Box2 of type Box
    Box Box3;               // Declare Box3 of type Box
    double volume = 0.0;     // Store the volume of a box here
    // box 1 specification Box1.setLength(6.0); Box1.setBreadth(7.0);
    Box1.setHeight(5.0);

    // box 2 specification Box2.setLength(12.0); Box2.setBreadth(13.0);
    Box2.setHeight(10.0);

// volume of box 1
volume = Box1.getVolume();
cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2
volume = Box2.getVolume();
cout << "Volume of Box2 : " << volume <<endl;

// Add two object as follows:
Box3 = Box1 + Box2;
```

```
  // volume of box 3
  volume = Box3.getVolume();
  cout << "Volume of Box3 : " << volume <<endl;

  return 0;
}
```

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

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

The unary operators operate on a single operand and following are the examples of Unary operators:

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

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

class Distance
{
  private:
    int feet;          // 0 to infinite int inches;          // 0 to 12
  public:
    // required constructorsDistance(){
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i){
      feet = f;
      inches = i;
    }
    // method to display distance void displayDistance()
    {
      cout << "F: " << feet << " I:" << inches <<endl;
    }
```

```
    // overloaded minus (-) operator
    Distance operator- ()
    {
      feet = -feet;
      inches = -inches;
      return Distance(feet, inches);
    }
};
int main()
{
  Distance D1(11, 10), D2(-5, 11);

  -D1;                  // apply negation
  D1.displayDistance();   // display D1

  -D2;                  // apply negation
  D2.displayDistance();   // display D2

  return 0;
}
```

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

```
F: -11 I:-10
F: 5 I:-11
```

Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

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

class Box
{
  double length;      // Length of a box double
  breadth;     // Breadth of a box double
  height;      // Height of a box
public:

  double getVolume(void)
  {
    return length * breadth * height;
```

```
}
      void setLength( double len )
      {
         length = len;
      }

      void setBreadth( double bre )
      {
         breadth = bre;
      }

      void setHeight( double hei )
      {
         height = hei;
      }
      // Overload + operator to add two Box objects. Box operator+(const Box& b)
      {
        Box box;
        box.length = this->length + b.length; box.breadth = this->breadth + b.breadth; box.height = this-
        >height + b.height; return box;}
   };
   // Main function for the program int main( )
   {
      Box Box1;               // Declare Box1 of type Box
      Box Box2;              // Declare Box2 of type Box
       Box Box3;               // Declare Box3 of type Box
      double volume = 0.0;     // Store the volume of a box here

      // box 1 specification Box1.setLength(6.0); Box1.setBreadth(7.0);
      Box1.setHeight(5.0);

      // box 2 specification Box2.setLength(12.0); Box2.setBreadth(13.0);
      Box2.setHeight(10.0);

      // volume of box 1
      volume = Box1.getVolume();
      cout << "Volume of Box1 : " << volume <<endl;

      // volume of box 2
      volume = Box2.getVolume();
```

```
        cout << "Volume of Box2 : " << volume <<endl;

        // Add two object as follows:
        Box3 = Box1 + Box2;


        // volume of box 3
  volume = Box3.getVolume();
  cout << "Volume of Box3 : " << volume <<endl;


  return 0;
}
```

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

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

The increment (++) and decrement (--) operators are two important unary operators available in C++.

Following example explain how increment (++) operator can be overloaded for prefix as well as postfix usage. Similar way, you can overload operator (--).

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


class Time
{
  private:
    int hours;           // 0 to 23 int minutes;         // 0 to 59
  public:
    // required constructors
    Time(){
      hours = 0;
      minutes = 0;
    }
    Time(int h, int m){
      hours = h;
      minutes = m;
    }
    // method to display time void displayTime()
    {
      cout << "H: " << hours << " M:" << minutes <<endl;
```

```
    }
    // overloaded prefix ++ operator
    Time operator++ ()
    {
      ++minutes;          // increment this object if(minutes >= 60)
      {
        ++hours;
        minutes -= 60;}
      return Time(hours, minutes);
    }
    // overloaded postfix ++ operator
    Time operator++( int )
    {
      // save the orignal value
      Time T(hours, minutes);
      // increment this object
      ++minutes;
      if(minutes >= 60)
      {
        ++hours;
        minutes -= 60;
      }
      // return old original value return T;
    }
};
int main()
{
  Time T1(11, 59), T2(10,40);
  ++T1;                // increment T1
  T1.displayTime();       // display T1
  ++T1;                // increment T1 again
  T1.displayTime();       // display T1


  T2++;                // increment T2
  T2.displayTime();       // display T2
  T2++;                // increment T2 again T2.displayTime();       // display T2 return 0;
}
```

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

```
 H: 12 M:0
H: 12 M:1H: 10 M:41
H: 10 M:42
```

Type Casting:-

12. [Typecasting] Write a program to demonstrate the typecasting of basic type to class type.

13. [Typecasting] Write a program to demonstrate the typecasting of class type to basic type.

14. [Typecasting] Write a program to demonstrate the typecasting of class type to class type.

Code for PROGRAM OF BASIC TO CLASS CONVERSION in C++ Programming

```
# include <iostream.h>
# include <conio.h>
# include <stdlib.h>

class date
{
   int mm,dd,yy;
   public:
   date()
   {
   }
   date(char dt[])
   {
      dd=getdt(dt,0); mm=getdt(dt,3); yy=getdt(dt,6);
   }

   int getdt(char *,int);
   void display();
};

main()
{
date d1; char dt[10]; clrscr();
cout<<"ENTER THE DATE IN DD/MM/YY FORMAT:->";
cin>>dt;
d1=date(dt);
d1.display();
}

int date::getdt(char dt[],int i)
{
   char temp[2]; temp[0]=dt[i]; temp[1]=dt[i+1]; temp[2]='\0';return atoi (temp);
}
void date::display()
{
  cout<<"DAY= "<<dd<<endl; cout<<"\nMONTH= "<<mm<<endl; cout<<"\nYEAR=
  "<<yy<<endl;
}
```

Code for Program of class to basic conversion in C++ Programming

```
# include <iostream.h>
# include <conio.h>
# include <stdlib.h>
class date
{
   int dd,mm,yy;
   public:
   date()
   {
   }
   date(int cdd,int cmm,int cyy)
   {
      dd=cdd; mm=cmm; yy=cyy;
   }
   operatorchar *();
};

main()
{
   clrscr();
   date d1,d2;
   int cmm,cdd,cyy;
   cout<<"\nENTER THE DAY:->";
   cin>>cmm;
   cout<<"\nENTER THE MONTH:->";
   cin>>cdd;
   cout<<"\nENTER THE YEAR:->";
   cin>>cyy; char dt[10]; d2.date(cdd,cmm,cyy); strcpy(dt,d2);
}

date::operatorchar *()
{
   char *str; char a[10]; str=a;
   int i=1;
   while(1){
      *str=dd/10+'0';
      str++;
      *str=dd%10+'0';
      str++;
      i++;
      if(i>3)
         break;
      *str='/'; str++; if(i==2)
                                                dd=mm;
      else dd=yy;
}
}
```

Code for Program of class to class conversion in C++ Programming

```
# include <iostream.h>
# include <conio.h>
class in1
{
   int code,items; float price; public:
```

```
  in1(int a,int b,int c)
  {
     code=a; items=b; price=c;
  }
  void putdata()
  {
     cout<<"CODE= "<<code<<endl; cout<<"ITEMS= "<<items<<endl; cout<<"VALUE=
     "<<price<<endl;
  }
  int getcode()
  {
     return code;
  }
  int getitems()
  {
     return items;
  }
  int getprice()
  {
     return price;
  }
  operatorfloat ()
  {
      return items*price;}
};

class in2
{
   int code; floatvalue; public: in2()
   {
      code=0;
      value=0;
   }
   in2(int x,float y)
   {
      code=x;
      value=y;
   }
   void putdata()
   {
      cout<<"CODE= "<<code<<endl;
      cout<<"VALUE= "<<value<<endl;
   }
   in2(in1 p)
   {
      code=p.getcode();
      value=p.getitems()*p.getprice();
   }
};

main()
{
clrscr();
   in1 s1(100,51,140.0);
   float tot_value;
   in2 d1; tot_value=s1; d1=in1(s1);
   cout<<"PRODUCT DETAILS INVENT-1 TYPES:->"<<endl;
```

```
    s1.putdata();
    cout<<"STOCK VALUE"<<endl;
    cout<<"VALUE= "<<tot_value<<endl;
    cout<<"PRODUCT DETAILS INVENT-2 TYPES:->"<<endl;
    d1.putdata();
}
```

Inheritance and Polymorphism:-

15. [Inheritance] Write a program to demonstrate the multilevel inheritance.

16. [Inheritance] Write a program to demonstrate the multiple inheritance.
17. [Inheritance] Write a program to demonstrate the virtual derivation of a class.

18. [Polymorphism] Write a program to demonstrate the runtime polymorphism.

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS- A mammal hence dog IS-A animal as well and so on.

Base & Derived Classes:

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows:

#include <iostream>

```
using namespace std;

// Base class class Shape
{
                                                    public:
    void setWidth(int w)
    {
      width = w;
    }
```

```
      void setHeight(int h)
      {
        height = h;
      }
   protected:int width;
   int height;
};

// Derived class
class Rectangle: public Shape
{
                                                    public:
   int getArea()
   {
     return (width * height);
   }
};

int main(void)
{
  Rectangle Rect;

  Rect.setWidth(5);
  Rect.setHeight(7);

  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;

  return 0;
}
```

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

```
Total area: 35
```

Access Control and Inheritance:

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Type of Inheritance:

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

- **Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance:** When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
- **Private Inheritance:** When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Multiple Inheritances:

A C++ class can inherit members from more than one class and here is the extended syntax:

```
class derived-class: access baseA, access baseB....
```

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example:

```
#include <iostream>

using namespace std;

// Base class Shape class Shape
{
   public:
      void setWidth(int w){
        width = w;
     }
     void setHeight(int h)
     {
        height = h;
     }
  protected:
     int width;
     int height;
};

// Base class PaintCost class PaintCost
{
                                                    public:
     int getCost(int area)
     {
        return area * 70;
     }
};

// Derived class
class Rectangle: public Shape, public PaintCost
{
                                                    public:
     int getArea()
     {
        return (width * height);
     }
};
```

```
int main(void)
{
   Rectangle Rect;
   int area;

   Rect.setWidth(5);
   Rect.setHeight(7);

   area = Rect.getArea();

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   // Print the total cost of painting
   cout << "Total paint cost: $" << Rect.getCost(area) << endl;

   return 0;
}
```

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

```
Total area: 35
Total paint cost: $2450
```

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes:

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

                                                class Shape {
   protected:
      int width, height;
   public:
```

```
    Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    int area()
    {cout << "Parent class area :" <<endl;
      return 0;
    }
};
class Rectangle: public Shape{
                                                    public:
    Rectangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
      cout << "Rectangle class area :" <<endl;
      return (width * height);
    }
};
class Triangle: public Shape{
                                                    public:
    Triangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
      cout << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};
// Main function for the program int main( )
{
  Shape *shape; Rectangle rec(10,7);
  Triangle  tri(10,5);

  // store the address of Rectangle shape = &rec;
  // call rectangle area.
  shape->area();
```

```
   // store the address of Triangle shape = &tri;
   // call triangle area. shape->area();
   return 0;
}
```

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

```
Parent class area
Parent class area
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this:

```
                                        class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      virtual int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result:

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.Virtual Function:

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

Pure Virtual Functions:

It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      // pure virtual function virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

Exception Handling:-

19. [Exception Handling] Write a program to demonstrate the exception handling.

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
  // protected code
}catch( ExceptionName e1 )
{
  // catch block
}catch( ExceptionName e2 )
{
  // catch block
}catch( ExceptionName eN )
{
  // catch block
}
```

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

Throwing Exceptions:

Exceptions can be thrown anywhere within a code block using **throw** statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b)
{
  if( b == 0 )
  {
     throw "Division by zero condition!";
  }
  return (a/b);
}
```

Catching Exceptions:

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try
{
  // protected code
}catch( ExceptionName e )
{
 // code to handle ExceptionName exception
}
```

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

```
try
{
  // protected code
}catch(...)
{
 // code to handle any exception
}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

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


double division(int a, int b)
{
```

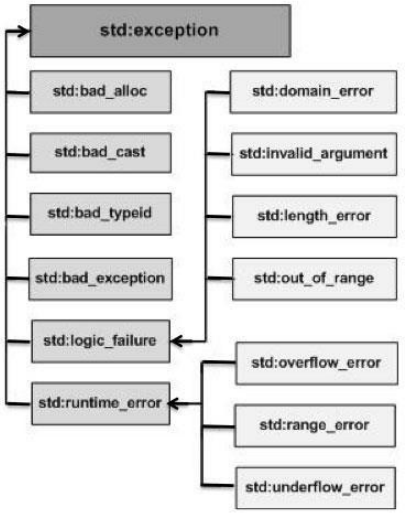
```
  if( b == 0 )
  {
     throw "Division by zero condition!";
  }
  return (a/b);
}

int main ()
{int x = 50;
  int y = 0;
  double z = 0;

  try {
    z = division(x, y);
    cout << z << endl;
  }catch (const char* msg) {
    cerr << msg << endl;
  }

  return 0;
}
```

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result:

Division by zero condition!

C++ Standard Exceptions:

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below:



Here is the small description of each exception mentioned in the above hierarchy:

Exception	Description
std::exception	An exception and parent class of all the standard C++ exceptions.
std::bad_alloc	This can be thrown by **new**.
std::bad_cast	This can be thrown by **dynamic_cast**.
std::bad_exception	This is useful device to handle unexpected exceptions in a C++ program
std::bad_typeid	This can be thrown by **typeid**.
std::logic_error	An exception that theoretically can be detected by reading the code.
std::domain_error	This is an exception thrown when a mathematically invalid domain is used
std::invalid_argument	This is thrown due to invalid arguments.
std::length_error	This is thrown when a too big std::string is created
std::out_of_range	This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[]().
std::runtime_error	An exception that theoretically can not be detected by reading the code.
std::overflow_error	This is thrown if a mathematical overflow occurs.
std::range_error	This is occured when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.

Define New Exceptions:

You can define your own exceptions by inheriting and overriding **exception** class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way:

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

struct MyException : public exception
{
  const char * what () const throw ()
  {
```

```
   return "C++ Exception";
                                                                  }
                                                                 };


int main()
{
  try
  {
    throw MyException();
  }
  catch(MyException& e)
  {
    std::cout << "MyException caught" << std::endl;
    std::cout << e.what() << std::endl;
  }
  catch(std::exception& e)
  {
    //Other errors
                                                                                  }}
```

This would produce the following result:

```
MyException caught
C++ Exception
```

Here, **what()** is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

Templates and Generic Programming:-

20. [Templates and Generic Programming] Write a program to demonstrate the use of function template.

21. [Templates and Generic Programming] Write a program to demonstrate the use of class template.

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes, let us see how do they work:

Function Template:

The general form of a template function definition is shown here:

```
template <class type> ret-type func-name(parameter list)
{
   // body of function
}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values:

```
#include <iostream>
#include <string>

using namespace std;
template <typename T>
inline T const& Max (T const& a, T const& b)
{
  return a < b ? b:a;
 }
 int main ()
{
```

```
   int i = 39;
   int j = 20;
   cout << "Max(i, j): " << Max(i, j) << endl;

   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;

   string s1 = "Hello";
   string s2 = "World";
   cout << "Max(s1, s2): " << Max(s1, s2) << endl;

  return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

Class Template:

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

```
template <class type> class class-name {
.
.
.
}
```

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>
```

```
using namespace std;

template <class T> class Stack { private:
   vector<T> elems;     // elements


 public:
   void push(T const&);  // push element void pop();               // pop element
   T top() const;            // return top element
   bool empty() const{       // return true if empty. return elems.empty();
   }
};

template <class T>
void Stack<T>::push (T const& elem)
{
   // append copy of passed element elems.push_back(elem);
}

template <class T>
void Stack<T>::pop ()
{
   if (elems.empty()) {
      throw out_of_range("Stack<>::pop(): empty stack");
   }
            // remove last elementelems.pop_back();
}

template <class T>
T Stack<T>::top () const
{
   if (elems.empty()) {
      throw out_of_range("Stack<>::top(): empty stack");
   }
```

```
            // return copy of last element
   return elems.back();
}

int main()
{
   try {
      Stack<int>         intStack;  // stack of ints
      Stack<string> stringStack;    // stack of strings

      // manipulate int stack intStack.push(7);
      cout << intStack.top() <<endl;

      // manipulate string stack stringStack.push("hello");
      cout << stringStack.top() << std::endl;
      stringStack.pop();
      stringStack.pop();
   }
   catch (exception const& ex) {
      cerr << "Exception: " << ex.what() <<endl;
      return -1;
   }
}
```

If we compile and run above code, this would produce the following result:

```
7
Hello
Exception: Stack<>::pop(): empty stack
```

File Handling:-

22. [File Handling] Write a program to copy the contents of a file to another file byte by byte. The name of the source file and destination file should be taken as command-line arguments,

23. [File Handling] Write a program to demonstrate the reading and writing of mixed type of data.

24. [File Handling] Write a program to demonstrate the reading and writing of objects.

So far, we have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

How to read and write from a file? This requires another standard C++ library called **fstream**, which defines three new data types:

Data Type	Description
ofstream	This data type represents the output file stream and is used to create files and to write information to files.
ifstream	This data type represents the input file stream and is used to read information from files.
fstream	This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File:

A file must be opened before you can read from it or write to it. Either the **ofstream** or **fstream** object may be used to open a file for writing and ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Mode Flag	Description
ios::app	Append mode. All output to that file to be appended to the end.

ios::ate	Open a file for output and move the read/write control to the end of the file.
ios::in	Open a file for reading.
ios::out	Open a file for writing.
ios::trunc	If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case it already exists, following will be the syntax:

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows:

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically closes flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File:

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File:

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read & Write Example:

Following is the C++ program which opens a file in reading and writing mode. After writing information inputted by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen:

```
#include <fstream>
```

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

int main ()
{

  char data[100];

  // open a file in write mode. ofstream outfile;
  outfile.open("afile.dat");

  cout << "Writing to the file" << endl;
  cout << "Enter your name: ";
  cin.getline(data, 100);

  // write inputted data into the file.
  outfile << data << endl;

  cout << "Enter your age: ";
  cin >> data;
  cin.ignore();

  // again write inputted data into the file.
  outfile << data << endl;

  // close the opened file.
  outfile.close();

  // open a file in read mode. ifstream infile;
  infile.open("afile.dat");

  cout << "Reading from the file" << endl;
  infile >> data;

  // write the data at the screen. cout << data << endl;
```

```
// again read the data from the file and display it. infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();

   return 0;
}
```

When the above code is compiled and executed, it produces the following sample input and output:

```
$./a.out
Writing to the file Enter your name: Zara Enter your age: 9
Reading from the file
Zara
9
```

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

File Position Pointers:

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are:

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );
// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );
// position n bytes back from end of fileObject fileObject.seekg( n, ios::end );
// position at end of fileObject fileObject.seekg( 0, ios::end );
```