```
Function Body
}
```

A virtual function cannot be a static member since a virtual member is always a member of a particular object in a class rather than a member of the class as a whole.

```
class point {
int x;
int y;
public:
virtual static int length (); //error
};
int point: : length ( )
{
Function body
{
```

A virtual function cannot have a constructor member function but it can have the destructor member function.

```
class point {
int x;
int y;
public:
virtual point (int xx, int yy); // constructors, error
void display ( );
int length ( );
};
```

A destructor member function does not take any argument and no return type can be specified for it not even void.

```
class point {
int x;
int y;
public:
virtual _ point (int xx, int yy); //invalid
void display ( );
int length ( );
```

It is an error to redefine a virtual method with a change of return data type in the derived class with the same parameter types as those of a virtualll1ethod in the base class.

```
class base {
int x,y;
public:
```

```
virtual int sum (int xx, int yy ); //error
} ;
class derived: public base {
intz;
public:
virtual float sum (int xx, int yy);
};
```

The above declarations of two virtual functions are invalid. Even though these functions take identical arguments note that the return data types are different.

```
virtual int sum (int xx, int IT); //base class
virtual float sum (int xx, int IT); //derived class
```

Both the above functions can be written with int data types in the base class as well as in the derived class as

```
virtual int sum (int xx, int yy); //base class
virtual int sum (int xx, int yy); //derived class
```

Only a member function of a class can be declared as virtual. A non-member function (non-method) of a class cannot be declared virtual.

```
virtual void display () //error, nonmember function
{
Function body
}
```

9.5 LATE BINDING ABSTRACT CLASSES

Late binding means selecting functions during the execution. Though late binding requires some overhead it provides increased power and flexibility. The late binding is implemented through virtual functions as a result we have to declare an object of a class either as a pointer to a class or a reference to a class.

For example, the following shows how a late binding or run time binding can be carried out with the help of a virtual function.

```
class base {
private :
int x;
float y;
public:
virtual void display ( );
int sum ( );
};
class derivedD : public baseA
```

```
{
private:
int x;
float y;
public:
void display (); //virtual
int sum ( );
};
void main ( )
{
baseA *ptr;
derivedD objd;
ptr = &objd;
Other Program statements
ptr->display (); //run time binding
ptr->sum ( ); //compile time binding
}
```

Note that the keyword virtual is be followed by the return type of a member function if a run time is to be bound. Otherwise, the compile time binding will be effected as usual. In the above program segment, only the display ( ) function has been declared as virtual in the base class, whereas the sum ( ) is non-virtual. Even though the message is given from the pointer of the base class to the objects of the derived class, it will not access the sum ( ) function of the derived class as it has been declared as non-virtual. The sum ( ) function compiles only the static binding.

The following program demonstrates the run time binding of the member functions of a class. The same message is given to access the derived class member functions from the array of pointers. As function are declared as virtual, the C++ compiler invokes the dynamic binding.

```
#include<iostream.h>
#include<conio.h>
class baseA {
public:
virtual void display () {
cout<< "One \n";
}
};
class derivedB : public baseA
{
    public:
    virtual void display 0 {
    cout<< "Two\n"; }
};
class derivedC: public derivedB
{
    public:
    virtual void display ( ) {
    cout<< "Three \n"; }
};
void main ( ) {
    //define three objects
    baseA obja;
    derivedB objb;
    derivedC objc;
    base A *ptr [3]; //define an array of pointers to baseA
    ptr [0] = &obja;
    ptr [1] = &objb;
    ptr [2] = &objc;
    for (int i = 0; i <=2; i ++ )
    ptr [i]->display (); //same message for all objects
    getche ( );
}
```

Output

```
One
Two
Three
```

The program listed below illustrates the static binding of the member functions of a class. In program there are two classes student and academic. The class academic is derived from class student. The two member function getdata and display are defined for both the classes. *obj is defined for class student, the address of which is stored in the object of the class academic. The functions getdata ( ) and display () of student class are invoked by the pointer to the class.

```
#include<iostream.h>
#include<conio.h>
class student {
private:
int rollno;
char name [20];
public:
void getdata ( );
void display ( );
```

```
};
class academic: public student {
private:
char stream;
public:
void getdata ( );
void display ( );
} ;
void student:: getdata ( )
{
    cout<< "enterrollno\n";
    cin>> rollno;
    cout <<"enter    name \n";
    cin > > name;
}
void student:: display ( )
{
    cout<< "the student's roll number is "<<rollno<<"and name is"
<<name;
    cout<< endl;
}
void academic :: getdata ()
{
    cout<< "enter stream of a student? \n";
    cin >>stream;
}
void academic :: display () {
    cout<< "students stream \n";
    cout <<stream<< end!;
}

void main ( )
{
    student *ptr;
    academic obj;
    ptr=&obj;
    ptr->getdata ();
    ptr->display ();
    getche ( );
}
```

Output

```
enter rollno
25
enter name
raghu
the student's roll number is 25 and name is raghu
```

The program listed below illustrates the dynamic binding of member functions of a class. In this program there are two classes student and academic. The class academic is derived from student. Student function has two virtual functions getdata ( ) and display ( ). The pointer for student class is defined and object. for academic class is created. The pointer is assigned the address of the object and function of derived class are invoked by pointer to student.

```
#include<iostream.h>
#include<conio.h>
class student {
private:
introllno;
char name [20];
public:
virtual void getdata ( );
virtual void display ( );
};
class academic: public student {
private :
char stream[10];
public:
void getdata ( );
void display ( );
};
void s_dent:: getdata ( )
{
    cout<< "enter rollno\n";
    cin >> rollno;
    cout<< "enter name \n";
    cin >>name;
}
```

```
void student:: display ()
{
   cout<< "the student's roll number is "<<rollno<<"and name is"
<<name;
   cout<< endl;
}
void academic:: getdata ( )
{
   cout << "enter stream of a student? \n";
   cin>> stream;
}
void academic:: display 0
{
   cout<< "students stream \n";
   cout<< stream << endl;
}
void main ()
{

   student *ptr;
   academic obj;
   ptr = &obj;
   ptr->getdata ( );
   ptr->dlsplay ( );
   getche ( )
}
```

Output

```
enter stream of a student?
Btech
students stream
Btech
```

9.6 CONSTRUCTORS UNDER INHERITANCE

As we know, the constructors play an important role in initializing objects. We did not use them earlier in derived classes for the sake of simplicity. As long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to pass arguments to the base class constructor. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

It case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. Similarly, in a multi-level inheritance, the constructors will be executed in the order of inheritance. Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together when a derived class object is declared.

The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor.

Let us consider an example:

```
Derived (int a1, int a2, float b1, float b2, int d1):
        A(a1, a2), B (b1, b2)
    {
        d = d1;
    }
```

In the above constructor named derived, supplies the five arguments. The A(a1, a2) invokes the base constructor A( ) and B(b1, b2) invokes another base constructor B ( ). So the derived ( ) constructor has one argument d1 of its own the derived constructor contains two parts separated by a colon (:). The first part provides the declaration of the arguments that are passed to the derived constructor and the second part lists the function calls to the base constructors.

The constructor derived (.) may be invoked as follows:

```
......
derived obj (10, 20, 15.0, 17.5, 40);
......
```

The values are assigned to various parameters by the constructor derived ( ) as follows:

a1 → 10

a2 → 20

b1 → 15.0

b2 → 17.5

d1 → 40

The constructors for virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared. Any non-virtual bases are then constructed before the derived class constructor is executed.

Table 9.1: Execution of Base Class Constructors

Methods of Inheritance	Order of Execution
Class X: public y { };	Y(); base constructor X (); derived constructor

Class X: public y, public Z { };	Y(); base constructor (first)
	Z(); base constructor (second)
	X(); derived constructor

Class X: public Y, virtual public Z { };	Z(); virtual base constructor Y (); ordinary base constructor X(); derived constructor

Let us consider a program using constructors:

```
# include <iostream.h>
class A
{
    int a1;
    public:
      A (int x)
    {
      a1 = x:
      cout << "constructor of A\n";
    }
    void display (void)
    {
    cout << "a1 = " << a1 << "\n";
    }
}
class B
{
    float b1;
    public:
      B( float y)
      {
            b1 = y;
            cout << " constructor of B\n";
      }
      void display 1 (void)
      {
      cout << "b1 = " <<b1 << "\n";
```

```
        }
}
Class C: public B, public A
{
    int p, q;
    public c:
    c (int a, float b,. int c, int d): A(a), B(b)
    {
      p = c;
      q = d;
cout << "constructor of c \n";
}
void display 2 (void)
{
    count << "p = " << p << "\n";
    cout << "q = " << q << "\n";
}
}
main ( )
{
    c obj (10, 20.57, 40, 70);
    cout << "\n";
    obj . display ( );
    obj . display 1( );
    obj . display 2 ( );
}
```

The out of the above code would be as follows:

```
Constructor of B
Constructor of A
Constructor C
a1 = 10
b1 = 20.57
b = 40
q = 70
```

Note that B is initialized/called first, although it appears second in the derived constructor. This is because it has been declared first in the derived class header line. Also note that A(a1) and B(b1) are function calls. Therefore the parameter should not include types.

9.7 DESTRUCTORS UNDER INHERITANCE

If a derived class doesn't explicitly call a constructor for a base class, the default constructor for the parent class. In fact, the constructors for the parent classes will be called from the ground up. For example, if you have a base class Vehicle and Car inherits from it, during the construction of a Car, it becomes a Vehicle and then becomes a Car.

Destructors for a base class are called automatically in the reverse order of constructors.

One thing to think about is with inheritance you may want to make the destructors virtual:

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

This is important if you ever need to delete a derived class when all you ave is a pointer to the base class. Say you have a "Vehicle *v" and you eed to delete it. So you call:

delete v;

If Vehicle's destructor is non-virtual and this happens to actuall by a Car *", the destructor will not call Car:: ~ Car() - just Vehicle:: ~ Vehicle()

Destructor Execution Order

- As the objects go out of scope, they must have their destructors executed also, and since we didn't define any, the default destructors will be executed.
- Once again, the destruction of the base class object named unicycle is no problem, its destructor is executed and the object is gone.
- The sedan_car object however, must have two destructors executed to destroy each of its parts, the base class part and the derived class part. The destructors for this object are executed in reverse order from the order in which they were constructed.
- In other words, the object is dismantled in the opposite order from the order in which it was assembled. The derived class destructor is executed first, then the base class destructor and the object is removed from the allocation.
- Remember that every time an object is instantiated, every portion of it must have a constructor executed on it. Every object must also have a destructor executed on each of its parts when it is destroyed in order to properly dismantle the object and free up the allocation. Compile and run this program.

9.8 VIRTUAL DESTRUCTORS

When you may delete a derived object via a base pointer, virtual functions bind to the code associated with the class of the object, rather than with the class of the pointer/reference. When you say delete base ptr, and the base class has a virtual destructor, the destructor that gets invoked is the one associated with the type of the object *base ptr, rather than the one associated with the type of the pointer. This is generally a good thing.

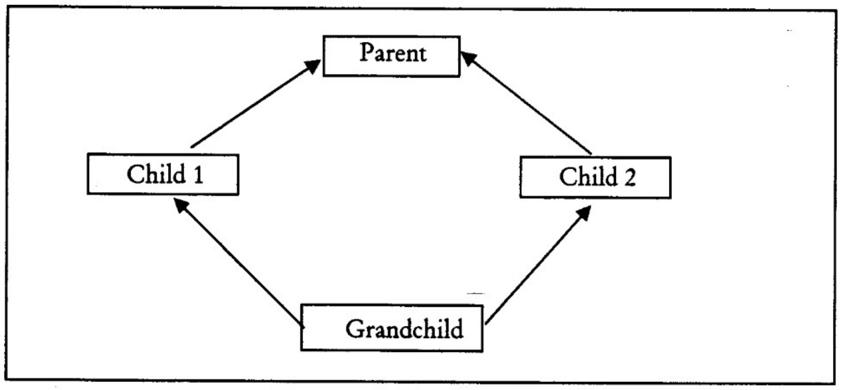
The destructor, ~ Thing (), does have a definition:

Virtual ~ Thing () { }

The definition is an empty function the destructor is virtual.

9.9 VIRTUAL BASE CLASSES

Consider the situation shown below:



Here, base class is parent, and two derived classes are child1 and child2. The fourth class Grandchild is derived from both child1 and child2.

In this arrangement a problem can arise if a member function in the Grandchild class wants to access data or functions in the parent class.

Eg: Ambiguous reference of base class

```
class parent
{
protected:
   int basedata;
};
 class child1,: public parent:
{ };
class child2,: Public parent:
{ };
class Grandchild: public child1,: Public child2:
{
public :
int getdata( )
{ return basedata;}    //Error: ambiguous
};
```

Error occurs when the getdata ( ) member function in Grandchild attempts to access basedata in parent. Why? When the child1 and child2 classes are derived from parents, each inherits a copy of parents: such copy is known to be subobject. Every subobject contains its own copy of parent's data, including basedata. When Grandchild refers to basedata, which of two copies will it access? The situation is ambiguous, and that's what the compiler reports.

To clear the ambiguity, are take child1 and child2 into virtual base classes, as shown by example virtbase:

e.g. Virtual base classes

```
class parent
{
protected:
int basedata;
};
class child1: virtual public parent //shares copy of parents {  };
class child2: virtual public parent //shares copy of parents {  };
class Grandchild: public child1, public child2
{
class Grandchild: public child1, public child2
{
public:
int getdata(  )
{ return basedata; } // ok: only one copy of parents
};
```

Check Your Progress

Fill in the blanks:

1. Polymorphism refers to the .... ability of a function to have different meanings in different contexts.
2. Late binding means selecting functions during the ....
3. It case of multiple inheritance, the base classes are constructed in the order in which they appear in the .... of the derived class.
4. The constructors for virtual base classes are ....before any non-virtual base classes.

9.10 LET US SUM UP

Polymorphism allows the program to use the exactly same function name with exactly same arguments in both a base class and its subclasses. It refers to the implicit ability of a function to have different meanings in different contexts.

Virtual functions do not really exist but it appears real in some parts of a program. To make a member function virtual, the keyword virtual is used in the methods while it is declared in the class definition but not in the member function definition.

Early binding refers to the events that occur at compile time while late binding means selecting function during the execution. The late binding is implemented through virtual function.

An abstract base class will not be used to create object, but exist only to act as a base class of other classes.

9.11 KEYWORDS

Virtual Function: Virtual functions, one of advanced features of OOP is one that does not really exist but it appears real in some parts of a program.

Late Binding: Selecting functions during the execution. Though late binding requires some overhead it provides increased power and flexibility.

9.12 QUESTIONS FOR DISCUSSION

1. What are the differences between virtual functions and virtual base class?
2. Write a program which uses a polymorphism with pointers.
3. Are virtual functions hierarchical? Give an example in support of your answer.
4. Write a program to use constructors and destructors in inheritance.
5. What are virtual destructors?

Check Your Progress: Model Answer

1. Implicit
2. Execution
3. Declaration
4. Invoked

9.13 SUGGESTED READINGS

Robert Lafore, *Object-oriented Programming in Turbo* C++, Galgotia Publications.

E Balagurusamy, *Object-Oriented Programming with* C++, Tata Mc Graw-Hill

Herbert Schildt, *The Complete Reference* C++, Tata Mc Graw Hill

LESSON

10

TEMPLATES AND EXCEPTION HANDLING

CONTENTS

10.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the concept of function template
- Define class template
- Describe the significance of exception handling

10.1 INTRODUCTION

The template is one of C++'s most sophisticated and high-powered features. Although not part of the original specification for C++, it was added several years ago and is supported by all modern C++ compilers. Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data upon which the function or class with several different types of data without having to explicity recode specific versions for each data type.

10.2 FUNCTION TEMPLATES

In C ++ when a function is overloaded, many copies of it have to be created, one for each data type it acts on. In the example of the max () function, which returns the greater of the two values passed to it this function would have to be coded for every data type being used. Thus, you will end up coding the same function for each of the types, like int, float, char, and double. A few versions of max () are

```
Int max ( int x, int y)
{
          return x > y ?  x : y
}
          char max ( char x, char y )
{
          return x > y ? > x : y
}
          double max (double x , double y)
{
          return x > y ? x : y
}
          float max ( float x, float y)
{
          return x > y ? x : y
}
```

Here you can see, the body of each version of the function is identical. The same code has to be repeated to carry out the same function on different data types. This is a waste of time and effort, which can be avoided using the template utility provided by C ++.

"A template function may be defined as an unbounded functions " all the possible parameters to the function are not known in advance and a copy of the function has to be created as and when necessary. Template functions are using the keyword, template. Templates are blueprints of a function that can be applied to different data types. The definition of the template begins with the template keyword. This is followed by a comma-separated list of parameter types enclosed within the less than (<) and greater than (>) signs.

Syntax of Template

Template < class type 1, type 2 ... >

Void function – name (type 2 parameter 1, type 1 parameter 2) {...}

Example

```
   Template < class X >
X min ( X a , X b )
   {
   return (a < b ) ? a : b ;
   }
```

This list of parameter types is called the formal parameter list of the template, and it cannot be empty. Each formal parameter consists of the keyword, type name, followed by an identifier. The identifier can be built-in or user-defined data type, or the identifier type. When the function is invoked with actual parameters, the identifier type is substituted with the actual type of the parameter. This allows the use of any data type. The template declaration immediately precedes the definition of the function for which the template is being defined. The template declaration, followed by the function definition, constitutes the template definition. The template of the max () function is coded below:

```
# include < iostream.h >
template < class type >
type max ( type x , type y)
{
    return x > y ? x : y ;
}
int main ( )
{
cout << " max ('A', 'a')          : " << max ('A', 'a') << endl ;
cout << " max (30,40 )            : " << max (30 , 40) << endl;
cout << " max (45.67F, 12.32 F)   : " << max (45.67F, 12.32F)<,endl;
return  0 ;
}
```

Output

```
max ( `A` , `a`)          : a
max ( 30 , 40 )           : 40
max ( 45.67F, 12 . 32F )  : 45 . 67
```

In the example, the list of parameters is composed of only one parameter. The next line specifies that the function takes two arguments and returns a value, all of the defined in the formal parameter list. See what happens if the following command is issued, keeping in mind the template definition for max ():

```
max ( a , b );
```

in which a and b are integer type variables. When the user invokes max ( ) , using two int values, the identifier, 'type', is substituted with int, wherever it is present. Now max ( ) works just like the function int max (int, int) defined earlier, to compare two int values. Similarly, if max ( ) is invoked using two double values, 'type' is replaced with 'double'. This process of substitution, depending on the parameter type passed to the function, is called template instantiation. The template specifies how individual functions will be constructed, given a set of actual types. The template facility allows the creation of a blueprint for a function like max ( ). The template or blueprint can then be instantiated for all data types, eliminating duplication of the source code. The identifier, 'type', can be used within the function body of a function that, otherwise, remains unchanged.

For example, the template for a function called square 9, which calculates the square of any number (int, float, or double type) passed to it can be given as:

```
# include < iostream.h >
template < class type >
type square ( type a)
{
   type b;
   b= a*a ;
   return b.;
}
   int main ( )
{
   cout << " square (25 )            : " << square ( 25) << endl;
   cout << " square 40               : " << square 40 << endl;
   return 0 ;
}
```

Output

```
Square ( 25 . 45F)  : 1125
Square 90           : 1600
```

Here is another example of the use of template function. Consider the following classes:

```
class IRON
{
private :
float density ;
public :
IRON ( ) {density = 8.9 }
Float Density ( ) { return density ;}
} ;
```

10.3 CLASS TEMPLATE

Template classes may be defined as the layout and operations for an unbounded set of related classes. Built in data types can be given as template arguments in the list for template classes.

Syntax

```
Template < class type 1 , ... >
Class class - name
{
public
type 1 var,
...
};
```

Suppose you want two different classes for the same data type this can be achieved through the following code:

```
# include < iostream.h >
template < class T, int z >
class W
{
  public :
  T a ;
  W ( T q )
  {
        a = z + q ;
        cout << " a = " << a << endl ;
  }
} ;
int main ( )
{
  W < int, 10 > one ( 100) ;           // Displays a = 110
  W < int, 10 > + twoptr = & one ;     // No object is created
  W < int, 10 + 3 > three = 200 ;      // Displays a = 230
  W < float, 40 > four = 100.45 ;      // Displays a = 140.45
  Return 0;
}
```

10.4 RULES FOR USING TEMPLATES

Listed below are the rules for using templates:

- The name of a parameter can appear only once within a formal parameter list. For example,

```
  Template <class type>
  Type max ( type a, type b)
  {
        return a > b ? a : b ;
  }
```

- The keyword, class must be specified before the identifier.
- Each of the formal parameters should form a part of the signature of the function. For example,

```
  Template < class T >
```

```
Void func ( T a )
{
            ......
}
```

- Templates cannot be used for all overloaded functions. They can be used only for those functions whose function body is the same and argument types are different.
- A built-in data type is not to be given as a template argument in the list for a template function but the function may be called with it. For example,

```
Template < class ZZ , int >    // wrong
Void fn (ZZ, int);
Template < class Z >           // right
Void fn (ZZ, int);
```

- Template arguments can take default values. The value of these arguments become constant for that particular instantiation of the template. For example,

```
Template < class X, int var = 69 >
Class Aclass
{
   public :
   void func ( )
   {
          X array [ var ] ;
   }
};
```

10.5 EXCEPTION HANDLING

We now know the syntactic errors and execution errors usually produce error messages when compiling or executing a program. Syntactic errors are relatively easy to find and correct, even if the resulting error messages are unclear. Execution errors, on the other hand, can be much more troublesome. When an execution error occurs, we must first determine its location (where it occurs) within the program. Once the location of the execution error has been identified, the source of the error (why it occurs) must be determined. Location of the execution error occurred often assists, however, in recognizing and correcting the error.

Closely related to execution errors are logical errors. Here the program executes correctly, carrying out the programmer's wishes, but the programmer has supplied the computer with instructions that are logically incorrect. Logical errors can be very difficult to detect, since the output resulting from a logically incorrect program may appear to be error-free. Moreover, logical errors are often hard to locate even when they are known to exist (as, for example).

Methods are available for finding the location of execution errors and logical errors within a program. Such methods are generally referred to as debugging techniques.

Check Your Progress

Fill in the blanks:

1. In C ++ when a function is overloaded, many .... of it have to be created, one for each data type it acts on.
2. Template classes may be defined as the layout and operations for an unbounded set of related ....
3. The name of a parameter can appear only once within a formal .... list.
4. .... errors are relatively easy to find and correct, even if the resulting error messages are unclear.

10.6 LET US SUM UP

In C++ when a function is overloaded, many copies of it have to be created, one for each data type it act on. A template function may be defined as an unbounded functions. The definition of the templates begins with the template keyword followed by a comma-separated list of parameter types enclosed within the less than (<) and greater than (>) signs.

Templates classes may be defined as the layout and operations for an unbounded set of related classes. While template functions and classes are usable for any data type, that would hold true only, as long as the body of functions or the class is identical throughout.

Specialization would consist of instantiating the templates definition for a specific data type if the templates instance does not exist, and the definition is not visible, generate an error.

Syntactic errors are relatively easy to find and correct, even if the resulting error messages are unclear. Execution errors, on the other hand, can be much more trouble some. Logical errors can be very difficult to detect, since that output resulting from a logically incorrect program may appear to be error-free. Errors isolation is useful for locating an error resulting in a diagnostic message.

10.7 KEYWORDS

Template: A template function may be defined as an unbounded functions. All the possible parameters to the function are not known in advance and a copy of the function has to be created as and when necessary.

Template Classes: Template classes may be defined as the layout and operations for an unbounded set of related classes.

Syntactic Errors: Syntactic errors are relatively easy to find and correct, even if the resulting error messages are unclear.

Logical Errors: Logical error can be very difficult to detect, since the output resulting from a logically incorrect program may appear to be error free.

10.8 QUESTIONS FOR DISCUSSION

1. What are templates?
2. Define template instantiation.
3. List down the rules for using templates.
4. How can you get a template function instantiated?

Check Your Progress: Model Answer

1. Copies
2. Classes
3. Parameter
4. Syntactic

10.9 SUGGESTED READINGS

Robert Lafore, *Object-oriented Programming in Turbo* C++, Galgotia Publications.

E Balagurusamy, *Object-Oriented Programming with* C++, Tata Mc Graw-Hill

Herbert Schildt, *The Complete Reference* C++, Tata Mc Graw Hill

LESSON

11

DATA FILE OPERATIONS

CONTENTS

11.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the concept of data file operations
- Discuss opening and closing of files
- Describe the stream state member functions
- Identify and explain the reading/writing a character from a file
- Discuss the binary file operations

- Explain the concept of class and file operations
- Discuss the array of class objects and file operations
- Identify the nested classes and file operations
- Explain the random access file processing

11.1 INTRODUCTION

Programs often input data from reading from a data file and output the result into another (or same) data file. This lesson will focus on issues related to accessing data from a data file through a C++ program.

11.2 DATA FILE OPERATIONS

Programs would not be very useful if they cannot input and/or output data from/to users. Some programs that require little or no input for their execution are designed to be interactive through user console – keyboard for input and monitor for output. However, when data volume is large it is generally not convenient to enter the data through console. In such cases data can be stored in a file and then the program can read the data from the data file rather than from the console.

The data file itself can exist in many forms. It may contain textual data in ASCII format or binary data. The data may be organized in fixed size record or may be in free form text. The various operations possible on a data file using C++ programs are:

- Opening a data file
- Reading data stored in the data file into various variables and objects in the program
- Writing data from a program into a data file
    - ❖ Removing the previously stored data in the file
    - ❖ Without removing the previously stored data in the file
        - ♦ At the end of the data file
        - ♦ At any other location in the file
- Saving the data file onto some secondary storage device
- Closing the data file once the ensuing operations are over
- Checking status of file operation

C++ treats each source of input and output uniformly. The abstraction of a data source and data sink is what is termed as stream. A stream is a data abstraction for input/output of data to and fro the program (Figure 11.1).

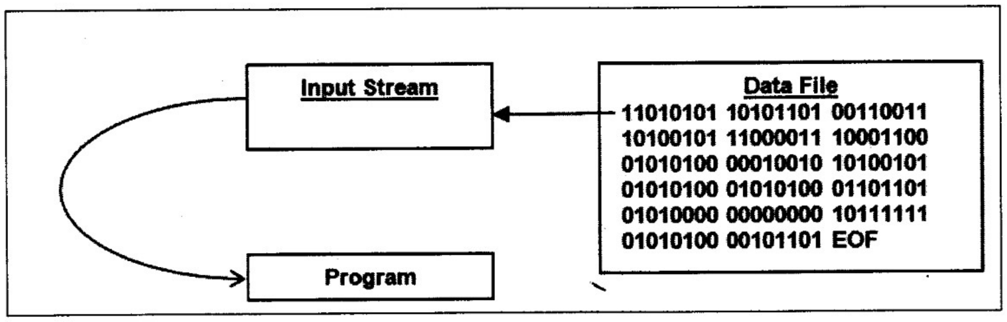


Figure 11.1 (a): Input Stream currently attached to a Data File

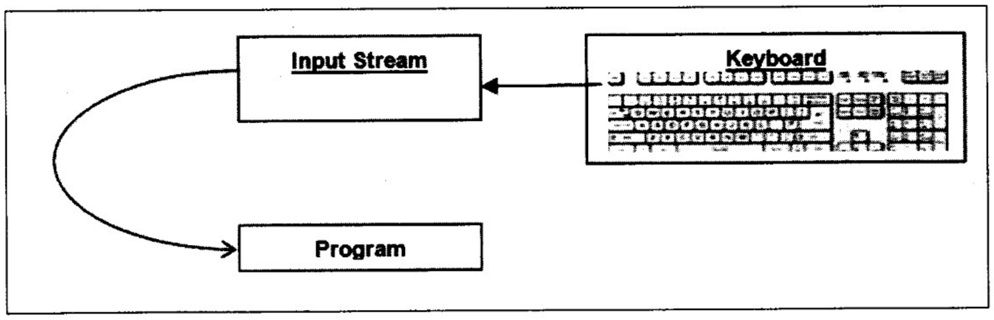


Figure 11.1 (b): Input Stream currently attached to Keyboard

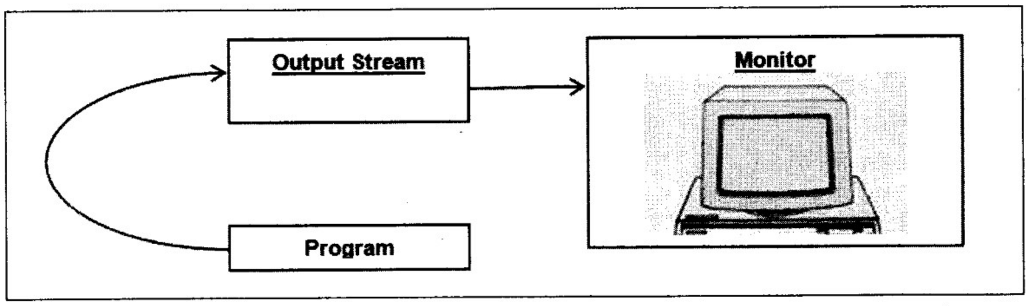


Figure 11.1 (c): Output Stream currently attached to Monitor

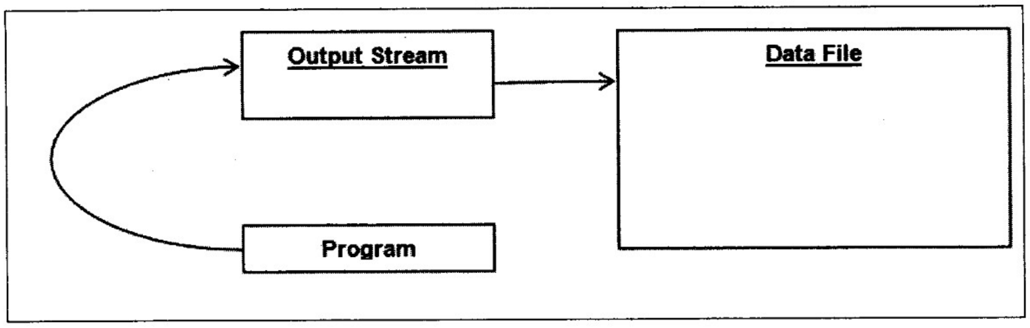


Figure 11.1 (d): Output Stream currently attached to a Data File