```
  cout<<"Before swapping\n";
  cout<<"x="<<x<<" y="<<y<<endl;
  swap(x,y);    //compiler generates swap(float&, float&);
                //and resolves the call
  cout<<"After swapping\n";
  cout<<"x="<<x<<" y="<<y<<endl;
}
/*End of swap02.cpp*/
```

Output

```
Before swapping
x=1.1 y=2.2
After swapping
x=2.2 y=1.1
```

Listing 9.6 Calling the template for the function 'swap' by passing floats

Objects of classes can also be passed to the function 'swap'. The compiler will generate an actual definition by replacing each occurrence of T by the name of the corresponding class.

```
/*Beginning of swap03.cpp*/
#include<iostream.h>
#include"swap.h"
#include"Distance.h"
void main()
{
  Distance d1(1,1.1),d2(2,2.2);
  cout<<"Before swapping\n";
  cout<<"d1="<<d1.getFeet()<<"'-"<<d1.getInches()<<"''\n";
  cout<<"d2="<<d2.getFeet()<<"'-"<<d2.getInches()<<"''\n";
  swap(d1,d2); //compiler generates swap(Distance&,
               //Distance&); and resolves the call
  cout<<"After swapping\n";
  cout<<"d1="<<d1.getFeet()<<"'-"<<d1.getInches()<<"''\n";
  cout<<"d2="<<d2.getFeet()<<"'-"<<d2.getInches()<<"''\n";
}
/*End of swap03.cpp*/
```

Output

```
Before swapping
d1=1'-1.1"
d2=2'-2.2"
```

```
After swapping
d1=2'-2.2"
d2=1'-1.1"
```

Listing 9.7 Calling the template for the function 'swap' by passing objects of the class 'Distance'

We must note the amount of effort saved in code development. Only one definition suffices for all possible types! Templates are a very handy tool provided by C++ for implementing code reusability.

Further, the compiler generates an actual function from a template only once for a given data type. For example, if the function 'swap' is called by passing integers for the first time, the compiler will generate a function definition from its template. Subsequent calls with the same data type will not generate the definition again. This is for the simple reason that the compiler first looks for an exact match to resolve a function call before looking for a template (the next paragraph explains this with the help of an example). If it finds an exact match, it does not look for a template. Since the first function call itself generates the function definition, subsequent calls do not do so.

Evidently, the entire definition of the function must appear in the header file. Otherwise, the compiler would not be able to generate the correct definition while compiling a user program in which the function template has been called.

The library programmer may like to put the definition of the template function in a library while keeping only the prototype in the header file. There is a keyword called `export` that is supposed to fulfill this need. However, not all compilers support this keyword.

It is sometimes necessary to override the function template by an actual function. In order to understand this, let us consider the template for a function to return the larger of the two arguments that are passed to it.

```
template <class T>
T& larger(const T& a, const T& b)
{
  return a>b? a:b;
}
```

Listing 9.8 Template for the larger function

This function works correctly if variables of ordinary data types such as `int` and `float` are passed to it. However, it does not work correctly if strings are passed to it.

```
char * s1="abcd", * s2="efgh";
char * s3=larger(s1,s2); //compiler generates larger(const
                         //char *&, const char *&); and
                         //resolves the call
```

Listing 9.9 Calling the larger function by passing strings

We notice that, during execution, the 'larger(char *&, char *&)' function in Listing 9.9 compares only the addresses of the two strings and not their contents! This is certainly not wanted. For this special case, we would like a special version of the function 'larger' for the character strings to execute. It is precisely a special version of the function 'larger' that we would define along with the template.

```
char * larger(char * a, char * b)
{
  return strcmp(a,b) > 0 ? a : b;
}
```

Listing 9.10 Overriding the template for the function 'larger'

Now, if the function 'larger' is called by passing two strings, the function in Listing 9.9 will be called while the template will be ignored. Function templates can be overloaded.

```
template <class T>
void display(const T & a)
{
  cout << a << endl;
}

template <class T>
void display(const T & a, const int n)  //overloaded
                                        //version of display()
{
  int ctr;
  for(ctr=0;ctr<n;ctr++)
    cout << a << endl;
}
```

Listing 9.11 Overloading a function template

The output of

```
display("C++");
```

will be

```
C++
```

But the output of

```
display("Hello",3);
```

will be

```
Hello
Hello
Hello
```

More than one generic type can also be mentioned in the template definition.

```
template <class T, class U>
void f1(const T & a, const U & b)
{
  //statements
}
```

Listing 9.12 More than one generic type in a function template

Here we should go back to Chapter 8 on Operator Overloading. It was mentioned that the need to make objects of a class capable of being used in function templates necessitates the overloading of operators for the class. We may look at the template for the function 'larger'. The 'greater than' operator is embedded within its definition. When objects of a certain class are passed as parameters to it, the 'greater than' operator will attempt to compare them. If this operator has not been overloaded for the class, the compiler will immediately report an error. Thus, in order to take advantage of the template, the 'greater than' operator should be overloaded for the class.

9.3 Class Templates

The need for class templates is similar to the need for function templates. The need for generic classes (Queue, Stack, Array, etc.) that handle data of different types is felt frequently. Let us consider the following set of three classes whose member functions have similar definitions, the mere difference being the type of the private data members upon whom they operate.

```
class X_for_int
{
    int val;
  public:
    void f1(const int &);
    void f2(const int &);
    /*
      rest of the class X_for_int
    */
};

class X_for_char
{
    char val;
  public:
    void f1(const char &);
    void f2(const char &);
    /*
      rest of the class X_for_char
    */
};

class X_for_string
{
    string val;
  public:
    void f1(const string &);
    void f2(const string &);
    /*
      rest of the class X_for_string
    */
};
```

Listing 9.13 Classes with similar definition

The classes 'X_for_int', 'X_for_char', and 'X_for_string' defined in Listing 9.13 are similar in every respect except for the type of their data members. As expected, the presence of three different classes that are different only in the data type of the data members upon whom their member functions work creates huge difficulties in code maintenance. Any change in one of the classes will have to be replicated in all of the others. This situation certainly demands the creation of a template class. Such a template class can be created as follows:

```
template<class T>
class X
{
    T val;
  public:
    void f1(const T &);
    void f2(const T &);
    /*
      rest of the class X
    */
};
```

Listing 9.14 A class template

The definition of the template class begins with the keyword `template`. This is followed by the list of type and non-type template arguments enclosed in angular brackets. 'Type' template arguments are those that represent a data type. An actual built-in or user-defined type replaces them when an object is declared. Each type template argument is preceded by the keyword `class`. 'Non-type' template arguments are variables of built-in or user-defined type. Actual constant values are passed for these non-type template arguments. The data type precedes each non-type template argument. Thereafter, the class is defined using the usual syntax.

Member functions of class templates are defined as follows:

```
template<class T>
void X<T> :: f1(const T & p)
{
  /*
    definition of the function
  */
}
```

Listing 9.15 Defining the member function of a class template

Member functions of a template class are defined in the same way as the template class itself. The definition begins with the `template` keyword. This is followed by the list of type and non-type template arguments enclosed in angular brackets. Each type template argument is preceded by the keyword `class`; each non-type template argument is preceded by its data type. Thereafter, the function is defined using the usual syntax except for one important difference. The class name given before the scope resolution operator is followed by the names of all template arguments enclosed in angular brackets.

Objects of this template class can be declared as follows:

```
X<int> intObj;
```

While declaring the object, the class name is followed by the type and non-type template parameter(s) enclosed in angular brackets. This is followed as usual by the name of the object itself. When the compiler sees the declaration of the object, it replaces each occurrence of the template argument by the template parameter in the definition of the class template and generates a separate class. In the preceding case, each occurrence of the token T in the class X will be replaced by the keyword `int`.

Objects of template classes, once declared, can be used just like any other object.

```
X<int> intObj01,intObj02;
intObj01.f1(intObj02);
```

The compiler generates the exact definition of a class from a given class template once only for each data type. For example, if two objects of the template class X are declared with the data type `int`, the compiler will generate the exact definition for the first object only.

```
X<int> intObj01;     //definition generated and used
X<int> intObj02;     //no definition generated
```

As in the case of non-member function templates, member functions of class templates are also defined in the header files themselves.

The section on Standard Template Library, which follows this section, has many instructive and practical examples of built-in class templates that are provided by all standard C++ compilers. Before moving on to that section, let us have a look at some fine points on class templates.

- A template class can take more than one template type argument The following listing (Listing 9.16) illustrates this.

```
template<class T, class U>
class X
{
    T val1;
    U val2;
    /*
      rest of the class X
    */
};
```

Listing 9.16 More than one template type argument in a class template

- A template class can take a non-type template argument. Listing 9.17 illustrates this.

```
template<class T, int v>
class X
{
    T val1;
    /*
      rest of the class X
    */
};
```

Listing 9.17 A non-type template argument in a class template

While declaring an object of such a class, a data type will be passed as a parameter for the type template argument. However, an actual value will be passed for the non-type template argument.

```
X<int,5> intObj;
```

- The name of the template argument cannot be used more than once in the template class's list of template arguments. The following listing (Listing 9.18) illustrates this.

```
template<class T, class T>    //ERROR: duplicate name in
                              //parameter list!
class X
{
    /*
      definition of class X
    */
};
```

Listing 9.18 Error due to identical names of more than one type template arguments

- The same name for a template argument can be used in the list of template arguments of two different template classes. Listing 9.19 illustrates this.

```
template<class T>
class X
{
    /*
      definition of class X
    */
};

template<class T>    //OK: Same name T used in two different
                     //classes
class Y
{
    /*
      definition of class Y
    */
};
```

Listing 9.19 Same name can be used for a type template argument in more than one class template

- The name of a template argument need not be the same in the declaration and the definition of the template class. Listing 9.20 illustrates this.

```
template<class T>
class X;        //declaration

template<class U>    //OK: different name for the template
                     //argument in the
class X    //definition
{
    /*
      definition of class X
    */
};
```

Listing 9.20 Name of a type template argument can be different in a template class declaration and its definition

- Formal arguments of template functions can be objects of a template class. The following listing (Listing 9.21) illustrates this.

```
template<class T>
class X
{
    /*
      definition of class X
    */
};

template<class U>
void f1(X<U> v)
{
  /*
    definition of the function
  */
}
```

Listing 9.21 Formal argument of a template function can be the object of a template class

Nested Class Templates

Nested classes can be created for template classes in the same way as they are created for non-template classes.

```
template<class T>
class A
{
    class B
    {
        T x;    //enclosing template type can be used in the
                //nested class
        /*
          rest of the class B
        */
    };
  /*
    definition of the class A
  */
};
```

Listing 9.22 A nested template class

9.4 The Standard Template Library (STL)

Would it not be of use if C++ provided class templates for meeting common programming requirements? For example, it would be highly convenient to have a class template that enables us to create a linked list of objects of any type of our choice.

The standard implementation of C++ does provide a set of header files where a large number of useful class templates have been defined. These files contain definitions of the class templates, their member functions, and a number of global associated functions. The global associated functions implement commonly used algorithms. This library of class templates and their helper global functions is known as the Standard Template Library (STL).

A complete study of all of these templates is beyond the scope of this book. However, we will study the more important class templates in the next section. The commonly used member functions and the associated global functions are explained with the help of examples.

The *list* Class

The 'list' class is used to create sequential containers. Elements of the list are single objects.

Objects of the 'list' class are declared as follows:

```
list<char> clist;   //creating a list of characters
list<float> flist;  //creating a list of floats
list<int> ilist;    //creating a list of integers
```

For using the 'list' template class, the header file 'list' needs to be included in the source code.

```
#include<list>
```

The elements of a list occupy a non-contiguous memory. They are doubly-linked through a pair of pointers. One of the pointers points at the next element and the other points at the previous element of the list. This allows both forward and backward traversal.

The number of elements a list object would have can be specified at the time of declaration.

```
list<int> ilist(3); //list of integers with three initial
                    //elements.
```

A default value can be specified for these elements.

```
list<int> ilist(3, -1);  //list of integers with three
                         //initial elements each having -1.
```

A list can be created from an existing array. We can do this by passing a pointer that points at the first element of the array and a second pointer that points 1 past the last element of the array to be copied.

```
int iArr[6] = {0,1,2,3,4,5};  //an array with six elements
list<int> ilist(iArr, iArr+6);     //list also has six
                                   //elements with the same
                                   //values
```

Let us have a look at the important member functions of this class.

The list<>::push_front() Function

This function is used to insert elements at the beginning of the list.

```
list<int> ilist;
ilist.push_front(1);      //inserts 1 at the beginning of
                          //the list
ilist.push_front(2);      //inserts 2 at the beginning of
                          //the list … list becomes 2,1.
```

The list<>::push_back() Function

This function is used to insert elements at the end of the list.

```
list<int> ilist;
ilist.push_back(1); //inserts 1 at the end of the list
ilist.push_back(2); //inserts 2 at the end of the list …
                    //list becomes 1,2.
```

The list<>::pop_front() Function

This function is used to delete the first element of the list.

```
list<int> ilist;
ilist.push_back(1); //inserts 1 at the end of the list
ilist.push_back(2); //inserts 2 at the end of the list …
                    //list becomes 1,2.
ilist.push_back(3); //inserts 3 at the end of the list …
                    //list becomes 1,2,3.
ilist.pop_front();  //deletes the first element … list
                    //becomes 2,3.
```

The list<>::pop_back() Function

This function is used to delete the last element of the list.

```
list<int> ilist;
ilist.push_back(1); //inserts 1 at the end of the list
ilist.push_back(2); //inserts 2 at the end of the list …
                    //list becomes 1,2.
ilist.push_back(3); //inserts 3 at the end of the list …
                    //list becomes 1,2,3.
ilist.pop_back();   //deletes the last element … list
                    //becomes 1,2.
```

Traversing a List using the Iterator

An iterator enables us to traverse the list elements in sequence. The following lines of code illustrate the syntax used for its declaration and its use.

```
list<int> ilist;
ilist.push_back(1);
ilist.push_back(2);
ilist.push_back(3);
list<int>::iterator iter=ilist.begin();       //iter points
                                              //at the first
                                              //element of
                                              //the list

for(;iter!=ilist.end();++iter)
  cout<<*iter<<endl;      //an iterator can be dereferenced
                          //just like a pointer
```

The 'list::begin()' function returns an iterator that points at the first element of the list. The 'list::end()' function returns an iterator that points 1 past the last element of the list. The 'increment' operator advances the iterator to point at the next element of the list. The 'indirection' operator (*) returns the value of the element pointed at by the iterator.

The list<>::insert() Function

This function enables a random insertion into a list.

```
list<int> ilist;
ilist.push_back(1); //inserts 1 at the end of the list
ilist.push_back(2); //inserts 2 at the end of the list …
                    //list becomes 1,2.
ilist.insert(ilist.begin(),-20);  //inserts -20 at the
                                  //beginning of the list
                                  //… list becomes
                                  //-20,1,2.
```

The 'list<>::insert()' function is used with the 'find()' function for random insertion into a list.

The find() Function

This global function searches specified values in lists. If the searched value is found in an element of the list, it returns an iterator to the element. Else, it returns the value of the 'list::end()' function.

The following program searches for the value '10' from the beginning of the list to the end. It inserts the value '–1' before '10' in the list, if the value is found. Else, it appends '–1' at the end of the list.

```
list<int>::iterator iter;
iter=find(ilist.begin(),ilist.end(),10);     //searching
                                        //from the first element to
                                        //the last element of the
                                        //list for the value 10
ilist.insert(iter,-1);
```

The list<>::size() Function

This function enables us to determine the number of elements currently in this list.

```
list<int> ilist;
ilist.push_back(1); //inserts 1 at the end of the list
ilist.push_back(2); //inserts 2 at the end of the list ...
                    //list becomes 1,2.
cout<<ilist.size()<<endl;     //outputs 2
```

The list<>::erase() Function

This function enables random deletion from the list. The iterator to the position of the element to be deleted is passed to the 'list::erase()' function.

Suppose we want to delete the element with value '19' from a list. We can use the 'find()' function to obtain the iterator to the element and pass it to the 'list::erase()' function.

```
iter=find(ilist.begin(),ilist.end(),19);     //obtaining an
                                       //iterator to the element
                                       //with value 19
if(iter!=ilist.end())                  //checking whether element
                                       //with value 19 exists or not
  ilist.erase(iter);                   //removing the element if
                                       //found
```

The list<>::clear() Function

This function erases all elements in a list.

```
list<int> ilist;
ilist.push_back(1);
ilist.push_back(2);
ilist.push_back(3);
cout<<ilist.size()<<endl;      //outputs 3
ilist.clear();                 //removes all elements of the
                               //list
cout<<ilist.size()<<endl;      //outputs 0
```

The list<>::empty() Function

This function is used to test whether a list is empty or not.

```
if(ilist.empty())
  //do something
else
  //do something else
```

Insertion into and deletion from an intermediate position in a list is efficient. This is because for such operations only the pointers of the affected element need to be reassigned.

On the other hand, random access to a particular element is inefficient. For traversing to the element that has our desired value, value of the pointer in each of the preceding elements has to be read starting from the first element since the elements are not in contiguous blocks of memory.

The *vector* Class

The 'vector' class is used to create sequential containers. Elements of the list are single objects.

The names of member functions of the 'vector' class are the same as those of the 'list' class. Global functions, such as the 'find()' function that works on objects of the 'list' class, have been overloaded to work upon objects of the 'vector' class too.

However, the layout of elements in a vector is completely different from that in a list. In a vector, unlike a list, elements are stored in contiguous blocks (just like an array).

For using the 'vector' template class, the header file 'vector' needs to be included in the source code.

```
#include<vector>
```

A vector does not actually regrow itself with each individual insertion. The amount of memory a vector captures is larger than the number of elements it actually stores. When this storage becomes full, it again regrows itself by a certain amount to accommodate the latest insertion. The amount by which a vector regrows differs from compiler to compiler.

This brings us to two important concepts about vectors, namely capacity and size.

Capacity is the total size of the block currently captured by a vector. Obviously, it is directly proportional to the total number of elements that can be inserted into the vector before it needs to regrow.

Size, on the other hand, is the number of elements actually stored in the memory block that has been captured by the vector. Obviously, size of a vector is always less than or equal to its capacity.

The 'vector' class has two functions that enable us to find the capacity and the size of the vector. These are 'vector::size()' and 'vector::capacity()'.

Insertion into and deletion from an intermediate position in a vector is inefficient. This is because for such operations all elements starting from the insertion point need to be pushed up or pushed down as the case may be.

On the other hand, random access to a particular element is efficient. For traversing to the element that has our desired value, only the internal iterator has to be incremented since the elements are in contiguous blocks of memory.

The *pair* Class

Objects of the 'pair' class represent a pair of values that may or may not be of the same type.

```
pair<string, int> player("Kasparov",1795);
```

The object 'player' in this statement may represent the number of games in our database that have been played by the player with name 'Kasparov' (the 'string' class is discussed later in this chapter). Obviously, we would like to create more variables of the same type later in the program. Using the keyword `typedef` allows us to do so.

```
typedef pair<string, int> Player;
Player kasparov("Kasparov",1795);
Player fischer("Fischer",2162);
Player karpov("Karpov",1525);
```

For using the 'pair' template class, the header file 'utility' needs to be included in the source code.

```
#include<utility>
```

The two elements of the objects of the 'pair' class can be accessed as first and second. For this, the member access operator can be used as follows:

```
cout<<"Number of games of "<<kasparov.first
    <<" are "<<kasparov.second;
```

The *map* Class

The 'map' class is used to create associative containers. Elements of the list are key/value pairs. The 'map' class does not allow duplicates.

For using the 'map' template class, the header file 'map' needs to be included in the source code.

```
#include<map>
```

Each record in the 'map' class is an object of the 'pair' class. The program (in Listing 9.23) illustrates all the important functionalities of the 'map' class.

```
/*Beginning of map.cpp*/
#include<iostream.h>
#include<map>
#include<utility>
void main()
{
  map<string, int> chessbase;
  typedef pair<string, int> Player;      //should be of the
                                         //same type as map

  Player kasparov("Kasparov",1795);
  Player fischer("Fischer",2162);
  Player karpov("Karpov",1525);

  chessbase.insert(kasparov); //inserting a record
  chessbase.insert(fischer);  //inserting another record
  chessbase.insert(karpov);   //inserting another record

  //The first member of each record is treated as the key.
  //The corresponding second member is the value and can be
  //retrieved as follows:
  cout<<"Number of games of Kasparov is: "
      <<chessbase["Kasparov"]<<endl;
  cout<<"Number of occurrences of Kasparov is: "
      <<chessbase.count("Kasparov")<<endl;
```

```
    //Using the subscript operator to query the value for a
    //key as above inserts it in the map!
    cout<<"Number of occurrences of Anand is: "
        <<chessbase.count("Anand")<<endl; //returns zero
    cout<<"Number of games of Anand is: "
        <<chessbase["Anand"]<<endl;  //returns zero … but a
                                     //record got added with
                                     //key as "Anand" and
                                     //value as zero because
                                     //value is of integer
                                     //type and zero is taken
                                     //as default value for
                                     //integers.
    cout<<"Number of occurrences of Anand is: "
        <<chessbase.count("Anand")<<endl; //return 1!!

    //An iterator can also be used. The iterator points at a
    //pair rather than a single value. The pair is returned
    //by the find function.
    map<string, int>::iterator iter;
    iter=chessbase.find("Tendulkar");
    cout<<"Number of occurrences of Tendulkar is: "
        <<chessbase.count("Tendulkar")<<endl;  //return 0

    if(iter!=chessbase.end())
      cout<<"Number of games of "<<iter->first<<" is: "
          <<iter->second<<endl;
    else
      cout<<iter->first<<" not found\n";
    cout<<"Number of occurrences of Tendulkar is: "
        <<chessbase.count("Tendulkar")<<endl;  //return 0 …
                                               //no new
                                               //record
                                               //inserted
}
/*End of map.cpp*/
```

Output

```
Number of games of Kasparov is: 1795
Number of occurrences of Kasparov is: 1
Number of occurrences of Anand is: 0
Number of games of Anand is: 0
Number of occurrences of Anand is: 1
Number of occurrences of Tendulkar is: 0
Tendulkar not found
Number of occurrences of Tendulkar is: 0
```

Listing 9.23 The 'map' class

The *set* Class

The 'set' class is used to create sequential containers. Elements of the list are single objects. A set stores a collection of keys in a sorted manner. The data itself serves as the keys to the set. The set contains the elements in a sorted fashion and duplicates are discarded during insertion.

For using the 'set' template class, the header file 'set' needs to be included in the source code.

```
#include<set>
```

An illustrative program (in Listing 9.24) follows:

```
/*Beginning of set.cpp*/
#include<set>
#include<string>
#include<iostream.h>
void main()
{
  set<char> set1;    //a set of characters

  string s1("I am indeed a cat. This is indeed a hat");
  cout<<s1<<endl;

  //Putting all the characters of the string s1 in the
  //set. Characters get automatically sorted while
  //duplicates get automatically rejected
  set1.insert(s1.begin(),s1.end());

  set<char>::iterator iter;
  for(iter = set1.begin(); iter!=set1.end(); iter++)
  {
    cout << *iter;   //outputting the set
  }
}
/*End of set.cpp*/
```

Output

```
I am indeed a cat. This is indeed a hat
.Itacdehimnst
```

Listing 9.24 The 'set' class

The *multimap* Class

The only difference between the 'map' and 'multimap' class is that while the 'map' class does not allow duplicate key values (it overrides the old value associated with a key), the 'multimap' class does allow duplicate key values. Therefore, the 'multimap' class does not support the 'subscript' operator.

For using the 'multimap' template class, the header file 'map' needs to be included in the source code.

```
#include<map>
```

The *multiset* Class

The only difference between the 'set' and 'multiset' class is that while the 'set' class does not allow duplicate key values (it overrides the old key value), the 'multiset' class does allow duplicate key values.

For using the 'multiset' template class, the header file 'set' needs to be included in the source code.

```
#include<set>
```

Summary

Templates enable generic programming. Templates are created for functions and classes that are similar to each other in every respect except for the type of data they work upon.

The compiler generates an actual function or a class from a template once and only once for a given data type.

The syntax for creating a template for a generic function is as follows:

```
template <class T, ...>
return_type function_name(T arg1, ...)
{
  //statements
}
```

The compiler generates an actual function definition from a function template when the function is called. The types of the template arguments in the function template are replaced by the data type of the parameters passed.

```
int x,y;
function_name(x,y); //definition function_name(int arg1,
                    //int arg2) generated
```

The syntax for creating a template for a generic class is as follows:

```
template <class T, ...>
class class_name
{
    T data_member_names;
    . . . .
    . . . .
  public:
    return_type function_name(parameter_names);
    . . . .
    . . . .
};
```

The syntax for defining member functions of template class is as follows:

```
template<class T, ...>
return_type class_name<T,...>::function_name(parameter_names)
{
  . . . .
  . . . .
}
```

The compiler generates an actual class definition from a class template when an object of the class is created. The types of the template arguments in the class template are replaced by the data type of the parameters passed to the object.

```
class_name<int> obj;      //actual definition of class
                          //class_name generated by
                          //replacing every occurrence of T
                          //by int.
```

A template class can take more than one template type argument.

A template class can take a non-type template type argument.

The name of the template argument cannot be used more than once in the template class's list of template arguments.

The same name for a template argument can be used in the list of template arguments of two different template classes.

The name of a template argument need not be the same in the declaration and the definition of the template class.

Formal arguments of template functions can be objects of template class.

Nested classes can be created for template classes in the same way as they are created for non-template classes.

The standard implementation of C++ provides a set of header files where a large number of useful class templates have been defined. These files contain definitions of the class templates, their member functions, and a number of global associated functions. The global associated functions implement commonly used algorithms. This library of class templates and their helper global functions is known as the Standard Template Library or STL.

The 'list' class is used to create sequential containers. Elements of the list are single objects.

For using the 'list' template class, the header file 'list' needs to be included in the source code.

```
#include<list>
```

The elements of a list occupy a non-contiguous memory. They are doubly linked through a pair of pointers. One of the pointers points at the next element and the other points at the previous element of the list. This allows both forward and backward traversal.

The 'vector' class is used to create sequential containers. Elements of the list are single objects.

In a vector, unlike a list, elements are stored in contiguous blocks (just like an array).

For using the 'vector' template class, the header file 'vector' needs to be included in the source code.

```
#include<vector>
```

Objects of the 'pair' class represent a pair of values that may or may not be of the same type.

For using the 'pair' template class, the header file 'utility' needs to be included in the source code.

```
#include<utility>
```

The 'map' class is used to create associative containers. Elements of the list are key/value pairs. The 'map' class does not allow duplicates.

For using the 'map' template class, the header file 'map' needs to be included in the source code.

```
#include<map>
```

Each record in a 'map' class is an object of the 'pair' class.

The 'set' class is used to create sequential containers. Elements of the list are single objects. A set stores a collection of keys in a sorted manner. The data itself serves as the keys to the set. The set contains the elements in a sorted fashion and duplicates are discarded during insertion.

For using the 'set' template class, the header file 'set' needs to be included in the source code.

```
#include<set>
```

The only difference between the 'map' and 'multimap' class is that while the 'map' class does not allow duplicate key values (it overrides the old value associated with a key), the 'multimap' class does allow duplicate key values.

For using the 'multimap' template class, the header file 'map' needs to be included in the source code.

```
#include<map>
```

The only difference between a 'set' and 'multiset' class is that while the 'set' class does not allow duplicate key values (it overrides the old key value), the 'multiset' class does allow duplicate key values.

For using the 'multiset' template class, the header file 'set' needs to be included in the source code.

```
#include<set>
```

Key Terms

function templates

class templates

STL

- list class
- vector class
- pair class
- map class
- set class
- multimap class
- multiset class

Exercises

1. What are function templates? What is the need for function templates? How are they created?
2. When and how does the C++ compiler generate an actual function definition from its template?
3. How is a function template overridden for a specific data type?

4. What are class templates? What is the need for class templates? How are they created?
5. When and how does the C++ compiler generate an actual class definition from its template?
6. State true or false.
   (i) The compiler generates an actual function definition from a function template only once for the same type of parameters.
   (ii) Function templates cannot be overloaded.
   (iii) A template class cannot take a non-type template argument.
   (iv) The name of a template argument need not be the same in the declaration and the definition of the template class.
7. What is the Standard Template Library? Name some of the template classes that are available in the STL.
8. Create a template for the bubble sort function.
9. Create a template for the 'Array' class.
10. Write a program that will show the following menu to the user:
    (i) Insert an integer at the end of the list
    (ii) Insert an integer at the beginning of the list
    (iii) Insert an integer before a specified integer in the list
    (iv) Delete the first integer from the list
    (v) Delete the last integer from the list
    (vi) Delete a specified integer from the list
    (vii) Display the list of integers
    (viii) Save the list of integers
    (ix) Quit

    Implement the above menu by using the 'list' class of the STL.
11. Assume that the user has used the program in Exercise 10 to save a list of integers (with plenty of duplicates) in a file. Declare a vector that would contain all positions of a given integer in the file. Suppose the contents of the file are:

```
21
19
3254
937
19
19
4253
335
19
9825
19
```

The vector for the integer 19 would contain the elements 2, 5, 6, 9, and 11. Write a code to populate the vector.

12. Create a 'pair' class that has the integer whose positions are to be stored as its first member and the vector that contains these positions as its second member. Rewrite the program in Exercises 10 and 11 to create such a pair object and assign 19 to its first member and a vector of its position as the second member.
13. Create a 'map' of two integers. The first member of the 'map' would represent a number that has been found in the file. The second member would represent the last position of the integer in the file. Write code to populate this map by the integers and their last positions in the file.
14. Declare a set of integers at the beginning of the program that you have written for Exercise 10. Keep updating the set as the integers are inserted into or deleted from the list.

Chapter 10

Exception Handling

OVERVIEW

This chapter deals with exception handling. The benefits of exception handling and the much-needed protocol it establishes between the library and its applications are discussed. The chapter begins with a critical study of the C-style solution to the problem of exception handling. It then elucidates the use and mechanism of exception handling (the try-throw-catch mechanism). The need to throw class objects, the method of accessing members of thrown objects, and the use of nested exception classes are also discussed. The chapter concludes with a study of the limitations of exception handling.

10.1 Introduction

Let us begin by assuming the role of a library programmer. While defining non-member or member functions, we face situations where the function may or may not be able to execute further. For example, we write a statement to divide one double type variable with another. Before this statement executes, we want to ensure that the denominator is not zero. We want to prevent the function from executing further if denominator is zero. This is only one of the conditions under which we want to prevent the further execution of the function. More such conditions exist (the function tries to open an unavailable file or requests more memory than is available). We know fully well the conditions under which the function should be aborted. However, we cannot decide the appropriate handling strategy. *While the library function can easily detect error conditions, it cannot decide upon an appropriate handling strategy.*

Now, let us assume the role of the application programmer. While calling a function, we should not be burdened with the task of detecting each error in the parameters that we pass to the functions we call. On the other hand, only we can decide what action should be taken whenever a particular error condition is met by the function being called. *While the user of the library function cannot detect error conditions, it can decide upon an appropriate handling strategy.*

Exception handling allows the library to sense and dispatch error conditions, and the client to handle them. It is usual for the library to know how to detect errors without knowing the appropriate handling strategy. It is just as usual for the client programs to understand how to deal with errors without being able to detect them.

We may wonder why a library function does not simply terminate the program when it detects invalid data input. Why does the library function not return an error value? All these questions will be answered in this chapter. Superiority of exception handling mechanism of C++ over the C-style error handling will also be discussed.

10.2 C-Style Handling of Error-generating Code

Let us study a function 'hmean()' that takes two float type numbers as parameters and computes their harmonic mean.

```
float hmean(const float a, const float b)
{
  return 2.0*a*b/(a+b);
}
```

Listing 10.1 Function to compute harmonic mean

Clearly 'a' and 'b' should not be the negative of each other, else it would result in division by zero. Every effort should be put in to prevent the evaluation of the return expression and the consequent run-time error if 'a' and 'b' are the negative of each other.

There are three traditional C-style solutions to this problem.

- Terminate the program
- Check the parameters before function call
- Return a value representing an error

These methods are discussed below.

Terminate the Program

Let us look at this solution.

```
/*Beginning of hmean.h*/
float hmean(const float, const float);
/*End of hmean.h*/

/*Beginning of hmean.cpp*/
#include"hmean.h"
#include<stdlib.h> // for abort()
float hmean(const float a, const float b)
{
  if(a==-b)
    abort();
  return 2.0*a*b/(a+b);
}
/*End of hmean.cpp*/

/*Beginning of hmeanmain.cpp*/
#include<iostream.h>
#include"hmean.h"
void main()
{
  float x,y,z;
  cout<<"Enter a number: ";
  cin>>x;
  cout<<"Enter another number: ";
  cin>>y;
  z=hmean(x,y);
  cout<<"Harmonic mean = "<<z<<endl ;
}
/*End of hmeanmain.cpp*/
```

Output

```
Enter a number: 10<enter>
Enter another number: -10<enter>
Abnormal program termination
```

Listing 10.2 Terminating the program when an error condition is met

This solution of terminating the program (as in Listing 10.2) is too extreme and drastic. The library function simply terminates the program on detecting an invalid input. Even if we do not provide the 'abort()' function, the OS anyway throws a similar or same error (depending upon the implementation) and terminates the program. This solution does not achieve anything tangible. The library user does not get a chance to take a corrective action of its choice. The library can and should do better.

Check the Parameters before Function Call

A program to prevalidate the function parameters is given below.

```
/*Beginning of hmean.h*/
float hmean(const float, const float);
/*End of hmean.h*/

/*Beginning of hmean.cpp*/
#include"hmean.h"
float hmean(const float a, const float b)
{
  return 2.0*a*b/(a+b);
}
/*End of hmean.cpp*/
/*Beginning of hmeanmain.cpp*/
#include<iostream.h>
#include"hmean.h"
void main()
{
  float x,y,z;
  while(1)
  {
    cout<<"Enter a number: " ;
    cin>>x;
    cout<<"Enter another number: " ;
    cin >>y ;
    if(x!=-y)
      break;
    cout<<"Invalid entry - enter again\n";
  }
  z=hmean(x,y) ;
  cout<<"Harmonic mean = "<<z<<endl ;
}
```

Output

```
Enter a number: 3<enter>
Enter another number: -3<enter>
Invalid entry – enter again
Enter a number: 2<enter>
Enter another number: 6<enter>
Harmonic mean = 3
```

Listing 10.3 Prevalidating function parameters to avoid error condition

This method relies upon the application programmer to prevalidate the data before passing them as parameters to the function call. However, it is not safe to rely upon the application programmer to know (or care) enough to perform such a check. A properly designed library function need not and should not burden the user with the task of checking the parameters for all invalid conditions.

Return a Value Representing an Error

Another approach is to use the function's return value to indicate a problem. Let us use a pointer argument or a reference argument to get a value back to the calling program and use the function's return value to indicate success or failure. By informing the calling function of the success or failure, we give the program the option of taking a suitable action of its choice. The program in Listing 10.4 shows an example of this approach. It redefines 'hmean()' function as an 'int' function whose return value indicates success or failure. It adds a third argument for obtaining the answer.

```
/*Beginning of hmean.h*/
int hmean(const float, const float, float const *);
/*End of hmean.h*/

/*Beginning of hmean.cpp*/
#include"hmean.h"
int hmean(const double a, const double b,double const * c)
{
  if(a==-b)
  {
    *c = 0;
    return 0; //return failure
  }
  else
```

```
    {
      *c=2.0*a*b/(a+b) ;
      return 1; //return success
    }
  }
  /*End of hmean.cpp*/

  /*Beginning of hmeanmain.cpp*/
  #include<iostream.h>
  #include"hmean.h"
  void main()
  {
    float x,y,z;
    int r;
    while(1)
    {
      cout<<"Enter a number: " ;
      cin >>x;
      cout<<"Enter another number:  " ;
      cin >>y;
      r=hmean(x,y,&z);
      if(r==1) //if success
        break;
      cout<<"Invalid entry - enter again\n" ;
    }
    cout<<"Harmonic mean = "<<z<<endl ;
  }
  /*End of hmeanmain.cpp*/
```

Output

```
Enter a number: 2<enter>
Enter another number: -2<enter>
Invalid entry – enter again
Enter a number: 2<enter>
Enter another number: 6<enter>
Harmonic mean = 3
```

Listing 10.4 Returning an error condition from the library function

The definition of the 'hmean' function as in Listing 10.4 does not burden the application program with the responsibility of prevalidating the parameters. It also allows the application program to take corrective action if it detects an error. Nevertheless, it still leaves the application program with the responsibility of detecting the error. The application program may bypass the test and use the value obtained by the third parameter! After all, the third parameter will certainly have some value or the other. The library function has no way of forcing the application program to take notice of the error condition!

To conclude, we should note that these C-style solutions are extreme in nature. They are either too strict (simply abort the program without allowing the application to take corrective action) or too lenient (merely return an error value without forcing the application program to take corrective action). What we need is a well-balanced solution by which the library function forces and at the same time allows its caller to take corrective action. Such a well-balanced solution is the exception-handling mechanism provided by C++.

10.3 C++-Style Solution—the try/throw/catch Construct

C++ offers the mechanism of exception handling as a superior solution to the problem of handling unexpected situations during run time. The following program illustrates the use of `try`, `throw`, and `catch` keywords for implementing exception handling. The advantages and limitations of this feature are discussed later.

```
/*Beginning of hmean.h*/
float hmean(const float, const float);
/*End of hmean.h*/

/*Beginning of hmean.cpp*/
#include"hmean.h"
float hmean(const float a, const float b)
{
  if(a==-b)
    throw "bad arguments to hmean()" ;
  return 2.0*a*b /(a+b) ;
}
/*End of hmean.cpp*/
/*Beginning of hmeanmain.cpp*/
#include<iostream.h>
#include"hmean.h"
void main()
{
  char choice='y' ;
  double x,y,z ;
  while(choice=='y')
  {
    cout<<"Enter a number: " ;
    cin>>x;
    cout<<"Enter another number: " ;
    cin >>y ;
    try
```

```
        {
          z=hmean(x,y);
        }
        catch(char * s)
        {
          cout<<s<<endl ;
          cout<<"Enter a new pair of numbers\n";
          continue;
        }
        cout<<"Harmonic mean of "<<x<< "and "<<y<< " is "<<z<<endl;
        cout<<"continue ? (y/n) ";
        cin>>choice;
      }
      cout<<"Bye\n";
    }
    /*End of hmeanmain.cpp*/
```

Output

```
Enter a number: 4 <enter>
Enter another number: -4 <enter>
bad arguments to hmean()
Enter a new pair of numbers
Enter a number: 2 <enter>
Enter another number: 6 <enter>
Harmonic mean of 2 and 6 is 8
continue ? (y/n) n <enter>
Bye
```

Listing 10.5 The try-throw-catch mechanism

Exception handling provides a way to transfer control from the library to the application. Handling an exception has three components. They are:

1. throwing an exception,
2. catching an exception with a handler, and
3. using a try block.

The `throw` keyword is used to throw an exception. It is followed by a value, such as character string or an object, indicating the nature of the exception. The library function notifies the user program about the error by throwing an exception.

The `catch` keyword is used to catch an exception. A catch-handler block begins with the keyword `catch` followed, in parentheses, by a type declaration indicating the type of exception that it catches. That, in turn, is followed by a brace enclosed block of code indicating the actions to take. The catch keyword, along with the exception types, is the point to which control should jump when an exception is thrown.

A try block encloses the block of code that is likely to throw an exception. Such a code generally consists of calls to library functions that are designed to throw errors in the manner described herein. One or more catch blocks follow the try block. The 'try' block is itself indicated by the keyword try followed by a brace—enclosed block of code indicating the code within which exception will be caught.

The try block looks like this:

```
try  // start of try block
{
  z=hmean(x,y);
}    // end of try block
```

If any statement in the try block causes an exception, the catch blocks after this block will handle the exception.

Exceptions are thrown as follows:

```
if(a==-b)
  throw "bad hmean() arguments : a = -b not allowed";
```

In this case, the thrown exception is the string 'bad hmean() arguments : a = -b not allowed'. The throw statement resembles the return statement because it terminates function execution. However, instead of merely returning control to the calling program, a throw causes the control to back up through the sequence of current function calls until it finds the try block. In this case, the throw passes program control back to 'main()'. There, the program looks for an exception handler (following the try block) that matches the type of exception thrown.

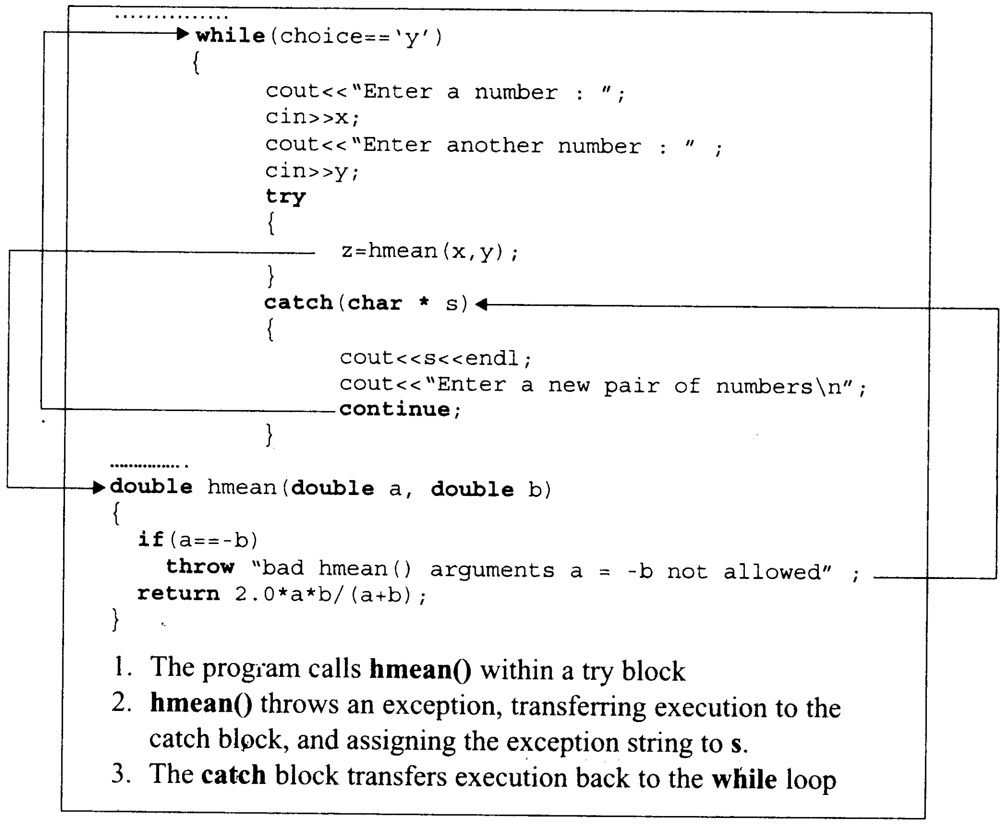
```
catch (char * s) // start of exception handler
{
  cout<<s<<"\n";
  cout<<"Enter a new pair of numbers : ";
  continue;
}                // end of handler
```

The keyword catch identifies the handler and the char * **s** means that this handler catches a string type exception. The thrown exception is assigned to 's'. Since the exception matches this handler, the program executes the code within the braces.

If a program completes executing statements in a try block without any exceptions being thrown, it skips the catch block or blocks after the try block and goes to the first statement following the handlers.

Let us follow the flow of control when the values '10' and '–10' are passed to the 'hmean()' function. The 'if' test succeeds and the exception (of char * type) is thrown. The 'hmean()'

function terminates. The control goes back to the point from where the 'hmean()' function was called and determines whether the call was embedded within a try block or not. It finds that the 'hmean()' function was called from the 'main()' function and that the call was embedded within a try block. The control then searches for a catch block that follows the try block and is of the matching `char *` type. The one and only catch block that follows the try block is of `char *` type. Therefore, the statements enclosed within it are executed. Diagram 10.1 illustrates this.



```
...............
while(choice=='y')
{
        cout<<"Enter a number : ";
        cin>>x;
        cout<<"Enter another number : " ;
        cin>>y;
        try
        {
                z=hmean(x,y);
        }
        catch(char * s)
        {
                cout<<s<<endl;
                cout<<"Enter a new pair of numbers\n";
                continue;
        }
..............
double hmean(double a, double b)
{
   if(a==-b)
     throw "bad hmean() arguments a = -b not allowed" ;
   return 2.0*a*b/(a+b);
}
```

1. The program calls **hmean()** within a try block
2. **hmean()** throws an exception, transferring execution to the catch block, and assigning the exception string to **s**.
3. The **catch** block transfers execution back to the **while** loop

Diagram 10.1 Flow of control when exceptions are thrown

In the introduction of this chapter, we had realized that an ideal solution to the problem of handling run-time errors should enable the library to sense and dispatch errors and the application to trap the dispatched error and take appropriate action. The exception-handling mechanism of C++ meets this requirement perfectly.

In order to appreciate the superiority of exception handling over the C-style solutions, we should keep the following two things in mind:

- It is necessary to catch an exception if it is thrown.
- When an exception is thrown, the stack is unwound.

It is Necessary to Catch Exceptions

The program terminates immediately if an exception thrown by a called function is not caught by the calling function. (A point to be borne in mind is that it is illegal to have a try block without a catch block). The program in Listing 10.6 is a case in point.

```
#include<iostream.h>

void abc(int);

void main()
{
  int i;
  abc(-1);
  for(i=1;i<=10;i++)
    cout<<i<<endl;
}

void abc(int x)
{
  if(x<0)
    throw "Invalid parameter";
}
```

Output

```
Abnormal program termination
```

Listing 10.6 Abnormal program termination due to uncaught exception

As we can observe in Listing 10.6, the remaining part of the 'main()' function after the call to the 'abc()' function does not execute. Instead, the program terminates. This happens because the call to the 'abc()' function has not been placed in a try block. Thus, when 'abc()' function throws an exception, there is no catch handler specified by the application programmer to execute a desirable piece of code. The program simply terminates with the default error message.

Thus, if the library programmer creates functions that throw exceptions, then the application programmer who uses the functions, is compelled to place the calls to such exception throwing library functions inside a try block and to provide suitable catch handlers.

Obviously, the library programmer should indicate the kinds of exceptions his/her function might throw. The list of exceptions a function throws is indicated in its prototype that is placed in the header file. The application programmer can find out what exceptions the library function throws by reading the header file. If a function, say 'abc()' function, throws exceptions of the `char *` type and `int` type and accepts an `int` type value as a parameter, then the function prototype should be as follows.

```
void abc(int) throw(char *,int);
```

Unwinding of the Stack

The throw statement unwinds the stack, cleaning up all objects declared within the try block by calling their destructors. Next, throw calls the matching catch handler, passing the parameter object.

The following program illustrates this fact.

```
#include<iostream.h>
class A
{
    int x ;
  public :
    A(int p)
    {
      x = p ;
      cout << "A  "<< x << endl ;
    }
    ~A()
    {
      cout << "~A  " << x << endl ;
    }
};

void abc();

void main()
{
  try
  {
    A A_main(1);
    abc();
  }
  catch(char * s)
  {
    cout<<s<<endl;
  }
}
```

```
void abc()
{
  A A_abc(2);
  throw "Exception thrown from abc()";
}
```

Output

```
A 1
A 2
~A 2
~A 1
Exception thrown from abc()
```

Listing 10.7 Unwinding of the stack due a thrown exception

As can be seen, throw destroys all objects from the point of throw until the try block. This action of the throw statement is clearly highlighted by the following program.

```
#include<iostream.h>
class A
{
    int x;
  public:
    A(int p)
    {
      x=p;
      cout<<"A "<<x<<endl;
    }
    ~A()
    {
      cout<<"~A  "<<x<<endl;
    }
};

void abc();
void def();
void ghi();

void main()
{
  try
  {
    A A_main(1);
    cout<<"calling abc()\n";
    abc();
  }
```

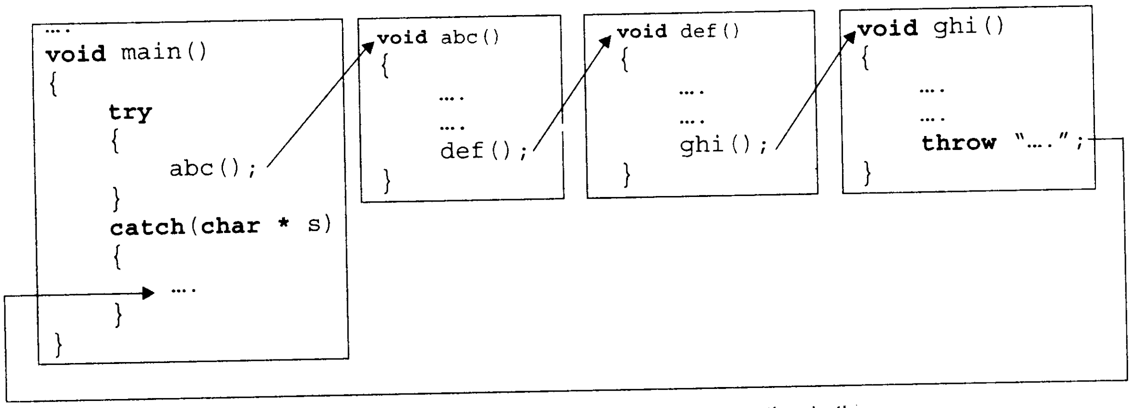
```
    catch(char * s)
    {
      cout<<s<<endl;
    }
}
void abc()
{
  A A_abc(2) ;
  cout<<"calling def()\n";
  def();
}

void def()
{
  A A_def(3) ;
  cout<<"calling ghi()\n" ;
  ghi();
}

void ghi()
{
  A A_ghi(4);
  throw "Exception from ghi()";
}
```

Output

```
A 1
calling abc()
A 2
calling def()
A 3
calling ghi()
A4
~A 4
~A 3
~A 2
~A 1
Exception from ghi()
```

Listing 10.8 Reversal of flow of control from the point of throw to the try block

In Listing 10.8, the try block does not contain a direct call to a function throwing an exception but it calls a function that throws an exception. Still, the control jumps from the function in which the exception is thrown to the function containing the try block and handlers. All local variables from the throw to the try block are destroyed.



```
....
void main()
{
    try
    {
        abc();
    }
    catch(char * s)
    {
        ....
    }
}
```

```
void abc()
{
    ....
    ....
    def();
}
```

```
void def()
{
    ....
    ....
    ghi();
}
```

```
void ghi()
{
    ....
    ....
    throw "....";
}
```

Diagram 10.2 Unwinding of stack when an exception is thrown

Need to Throw Class Objects

The problem with throwing values of fundamental data types is that the number of fundamental data types is limited. Thus, if two or more statements in a try block throw values of the same data type, then conflicts arise and it becomes difficult to detect the source of error in the catch block. The advantage with throwing objects of classes is that the library programmer can define any number of classes as exception classes. The following program illustrates this.

```
#include<iostream.h>
#include<math.h> //for sqrt()

class hmeanexcp{}; //an empty exception class
class gmeanexcp{}; //an empty exception class

double hmean(double,double);
double gmean(double,double);

void main()
{
  double x,y,z1,z2;
  char choice='y';
  while(choice=='y')
  {
    cout<<"Enter a number: ";
    cin>>x;
    cout<<"Enter another number: ";
    cin>>y;
    try
```

```
        {
          z1=hmean(x,y);
          z2=gmean(x,y);
        }
        catch(hmeanexcp)
        {
          cout<<"Exception error - a=-b not allowed\n";
          cout<<"Enter a fresh pair of numbers\n";
          continue;
        }
        catch(gmeanexcp)
        {
          cout<<"Exception error - a*b<0 not allowed\n";
          cout<<"Enter a fresh pair of numbers\n";
          continue;
        }
        cout<<"Harmonic mean = "<<z1<<endl;
        cout<<"Geometric mean = "<<z2<<endl;
        cout<<"Enter again ? (y/n)";
        cin>>choice;
    }
    cout<<"Bye\n";
}
double hmean(double a,double b)
{
    if(a==-b)
      throw hmeanexcp();//construct and throw objects!!
    return 2.0*a*b/(a+b);
}
double gmean(double a,double b)
{
    if(a*b<0)
      throw gmeanexcp();//construct and throw objects!!
    return sqrt(a*b);
}
```

Output

```
Enter a number: 10<enter>
Enter another number: -10<enter>
Exception error – a=-b not allowed
Enter a fresh pair of numbers
Enter a number: 10<enter>
Enter another number: -6<enter>
Exception error – a*b<0 not allowed
Enter a fresh pair of numbers
Enter a number: 16<enter>
Enter another number: 4<enter>
Harmonic mean = 6.4
```

```
Geometric mean = 8
Enter again? (y/n) n<enter>
Bye
```

Listing 10.9 Throwing objects of exception classes

We may note that it is not mandatory to declare an object of the exception class in the catch block. However, through the throw statements, we always throw objects (as has been done by calling constructors of the 'hmeanexcp' and 'gmeanexcp' classes in the 'hmean()' and 'gmean()' functions of Listing 10.9).

Accessing the Thrown Object in the Catch Block

If we declare an object of the exception class in the catch handler, then the thrown object gets copied into it. This object can then be accessed and used for further processing. The following program illustrates this.

```
include<iostream.h>
#include<math.h>
#include<string.h>

double hmean(double,double);
double gmean(double,double);

class hmeanexcp
{
    char cError[30];
  public:
    hmeanexcp(char * s)
    {
      strcpy(cError,s);
    }
    char * getcError()
    {
      return cError;
    }
};

class gmeanexcp
{
    char cError[30];
  public:
    gmeanexcp(char * s)
```

```
        {
          strcpy(cError,s);
        }
        const char * getcError()
        {
          return cError;
        }
    };

    void main()
    {
      double x,y,z1,z2;
      char choice='y';
      while(choice=='y')
      {
        cout<<"Enter a number: ";
        cin>>x;
        cout<<"Enter another number: ";
        cin>>y;
        try
        {
          z1=hmean(x,y);
          z2=gmean(x,y);
        }
        catch(hmeanexcp& e)
        {
          cout<<e.getcError()<<endl;
          cout<<"Enter a fresh pair of numbers\n";
          continue;
        }
        catch(gmeanexcp& e)
        {
          cout<<e.getcError()<<endl;
          cout<<"Enter a fresh pair of numbers\n";
          continue;
        }
        cout<<"Harmonic mean = "<<z1<<endl;
        cout<<"Geometric mean = "<<z2<<endl;
        cout<<"Enter again ? (y/n)";
        cin>>choice;
      }
      cout<<"Bye\n";
    }

    double hmean(double a,double b)
    {
      if(a==-b)
        throw hmeanexcp("Exception error - a=-b not allowed");
      return 2.0*a*b/(a+b);
    }
```

```
double gmean(double a,double b)
{
  if(a*b<0)
    throw gmeanexcp("Exception error - a*b<0 not allowed");
  return sqrt(a*b);
}
```

Output

```
Enter a number: 10<enter>
Enter another number: -10<enter>
Exception error – a=-b not allowed
Enter a fresh pair of numbers
Enter a number: 10<enter>
Enter another number: -6<enter>
Exception error – a*b<0 not allowed
Enter a fresh pair of numbers
Enter a number: 16<enter>
Enter another number: 4<enter>
Harmonic mean = 6.4
Geometric mean = 8
Enter again? (y/n) n<enter>
Bye
```

Listing 10.10 Accessing thrown objects

A temporary copy of the object to be thrown is created and thrown. Hence, the object in the catch handler refers to a copy of the thrown object. This is desirable because the thrown object disappears after the function from which it was thrown terminates. Thus, after the object is thrown, its three copies may be created—the object itself, its copy, and the object in the catch block. The object itself gets destroyed automatically when the function terminates. Therefore, we are left with two copies. In order to reduce this to one, we normally create a reference to the thrown object in the catch handler (as in the catch blocks of Listing 10.10).

Throwing Parameterized Objects of a Nested Exception Class

Let us have a look at the following program.

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

```
template<class T>
class vector
{
    T * v;
    Int size;
  public:
    class RangeError
    {
        char cError[30];
        int errorPos;
      public:
        RangeError(char * str,int p)
        {
          strcpy(cError,str);
          errorPos=p;
        }
        char * getcError()
        {
          return cError;
        }
        int getPos()
        {
          return errorPos;
        }
    };
    vector(int s)
    {
      v=new T[s];
      size=s;
    }
    ~vector()
    {
      delete[] v;
    }
    void setElement(T val,int p)
    {
      if(p>size-1 || p<0)
        throw RangeError("Out of range exception - could not    _
                          write", p);
      v[p]=val;
    }
    T getElement(int p)
    {
      if(p>size-1 || p<0)

        throw RangeError("Out of range exception - could not _
                          read",p);
        return v[p];
      }
};

void main()
{
  vector<int> int_vector(5);
  try
  {
    int_vector.setElement(3,5);
    cout<<int_vector.getElement(3)<<endl;
  }
  catch(vector<int>::RangeError& e)
  {
    cout<<e.getcError()<<" at position "<<e.getPos()+1<<endl;
  }

  vector<float> float_vector(6);
  try
  {
    float_vector.setElement(3.14,3);
    cout<<float_vector.getElement(6)<<endl;
  }
  catch(vector<float>::RangeError& e)
  {
    cout<<e.getcError()<<" at position "<<e.getPos()+1<<endl;
  }
}
```

Output

```
Out of range exception – could not write at position 6
Out of range exception – could not read at position 7
```

Listing 10.11 Nested exception class

We can define the exception class as a nested class of the class that throws it. This indicates the class originating an exception. It also prevents pollution of the global namespace. In the example in Listing 10.11, the class 'RangeError' has been declared within the 'vector' class. If the 'setElement()' function or the 'getElement()' function finds a bad subscript value, it throws an exception of type 'RangeError'. The handler for this exception looks like this

```
catch(vector<int>::RangeError& e) {...}
```

it may be noted that the nested exception class is public. This allows the catch block to have access to the type.

Catching Uncaught Exceptions

The C++ language supports a feature to catch exceptions that were raised in a try block but not caught by any of the catch blocks. The syntax of the catch construct to handle such exceptions is as follows.

```
catch(...)
{
     //action for handling an exception
}
```

The three dots in catch(...) indicate that it catches all types of exceptions. The following program illustrates the use of this catch block.

```
#include<iostream.h>

class Sugar{};
class Spice{};
class Tasteless{};
void abc(int);

void main()
{
  try
  {
    abc(-1);
  }
  catch(Sugar)
  {
    cout<<"Caught Sugar\n";
  }
  catch(Spice)
  {
    cout<<"Caught Spice\n";
  }
  catch(...)
  {
    cout<<"Unidentified object caught\n";
  }
}
void abc(int p)
{
  if(p<0)
    throw Tasteless();
}
```

Output

```
Unidentified object caught
```

Listing 10.12 Catching uncaught exceptions

10.4 Limitation of Exception Handling

The limitation of exception handling is that if a resource has been acquired (file has been opened, memory has been allocated dynamically in the heap area, etc.) and the statements to release the resource are after the throw statements, then the acquired resource may remain locked up. The following example illustrates this.

```
#include<iostream.h>
class A
{
  public:
    A()
    {
      cout<<"Constructor\n";
    }
    ~A()
    {
      cout<<"Destructor\n";
    }
};

void abc(int);

void main()
{
  try
  {
    abc(-1);
  }
  catch(char * s)
  {
    cout<<s<<endl;
  }
}

void abc(int p)
{
  A * Aptr = new A[2];
  if(p<0)
    throw "Invalid argument to abc()";
}
```

Output

```
Constructor
Constructor
Invalid argument to abc()
```

Listing 10.13 Dynamically allocated resources remain locked after the throw statement

In Listing 10.13, when the stack is unwound, memory occupied by the pointer 'Aptr' in 'abc()' function gets destroyed. However, the memory block at which the pointer points remains locked up. (This is evident from the fact that the destructor was not called for the objects created in the heap.)

In order to overcome this problem, classes whose objects function like pointers should be devised. Obviously, such objects will have pointers embedded in them. Memory will be allocated dynamically for these pointers during the lifetime of the objects. This memory can be deallocated through the destructor. Thus, when the object itself is destroyed, the memory locked up and referenced by the embedded pointer will also be destroyed.

Summary

Statements to detect conditions that prohibit further execution of the library function can and should be placed within the library function itself. Statements to take appropriate action when such conditions are detected can and should be placed in the functions that call these library functions.

Exception handling enables the library function to notify the detected invalid conditions to the user by using the throw statement. The program terminates prematurely if the application program ignores such notifications. Application program can catch such notifications in a try block and take appropriate action in a catch block.

Library functions can announce the list of all possible exceptions that they throw by enlisting them in their headers. Appropriately, the application program should place calls to these functions in a try block and append the try block with a series of catch blocks, one for each of the exceptions expected to be thrown.

Since the number of fundamental data types is limited, it is better to throw objects of exception classes created specifically for the purpose. These objects can be initialized with appropriate information before being thrown by the library functions. This information can then be accessed within the corresponding catch block of the application program.

Uncaught exceptions can be caught by the `catch(...)` {} construct. During unwinding of the stack, memory occupied by the objects themselves is destroyed. However, the memory acquired dynamically by the pointers embedded in these objects remains locked up. This is a limitation of exception handling.

Key Terms

exception handling

C-style solutions for exception handling try

catch

throw

catching uncaught exceptions

exception classes

unwinding of the stack

Exercises

1. What is exception handling? What is the need for exception handling?
2. What is the negative impact if the library programmer simply terminates an application upon detecting an error condition?
3. Which three keywords are provided by C++ for implementing exception handling?
4. What happens if the application does not catch the exception thrown by a library function?
5. Explain how the stack is unwound when an exception is thrown.
6. What is the need to throw class objects instead of values of fundamental types?
7. Why are nested exception classes needed?
8. How are uncaught exceptions caught?
9. What is the limitation of exception handling in C++?
10. Derive a class from another. Create two catch blocks—the first one for catching the base class type exception and the second one for catching the derived class type exception. Throw an exception of the base class type from the try block. Observe the result. Now, throw an exception of the derived class type from the try block. Observe and compare the results. Repeat the above two observations by reversing the sequence of the catch blocks. What do you conclude?
11. Add a function to the class 'String' that will return the character from the position that is passed as a parameter to it. If the position is out of bounds, the function should throw a user-defined exception.

Case Study

A.1 A Word Query System

Problem Statement

The word query system should allow us to determine whether a particular word exists in a given text file or not.

If the program finds the word being searched, it would display all the lines in which the word was found. The program would also display the number of occurrences of the word, the serial number of the line in the file and the position of the word in the line.

A Sample Run

The file having the following lines, written by the author about his favorite game, can be taken as input:

```
Chess is the most intellectual mind sport known to mankind.
A game of chess is a war of intelligence and a clash of
wills. It is a game of kings, queens, rooks, knights,
bishops and pawns. What appears to be a two-dimensional
black and white board of 64 squares is, for the chess
master, a multidimensional multicolored wonderland of
cunning strategy and brilliant tactics. Chess has a rich
and long history. Invented in India as a war game, it has
followers all over the world. Of all the sports, it has
perhaps the largest literature. To be a true master of the
game requires years of hard labor, study and practice. The
game has been played by kings and by commoners alike. A
regular practice of the game leads to better concentration
and an improved ability to deduce facts from logic.
```

A sample run of the program is as follows (we would implement a case sensitive search):

Please enter the word to be searched (enter blank to quit): **chess** *<enter>*

Number of occurrences of 'chess' = 2

```
(2,4) A game of chess is a war of intelligence and a clash of
(5,11) black and white board of 64 squares is, for the chess
Please enter the word to be searched (enter blank to quit): master <enter>
Number of occurrences of 'master' = 2
(6, 1) master, a multidimensional multicolored wonderland of
(10,9) perhaps the largest literature. To be a true master of the
Please enter the word to be searched (enter blank to quit): mind <enter>
Number of occurrences of 'mind' = 1
(1,6) Chess is the most intellectual mind sport known to mankind.
Please enter the word to be searched (enter blank to quit): golf <enter>
Number of occurrences of 'golf' = 0
Please enter the word to be searched (enter blank to quit): <enter>
Bye!
```

The Source Code

The program listing to implement the word query system as described in the Problem Statement follows:
(Please note that the code calls a few of the member functions of the library string class. These simple calls have been explained in the accompanying comments.)

```
/*Beginning of textQuerySearch.cpp*/
#include<string>//the library string class
#include<vector>
#include<fstream.h>
#include<map>

using namespace std;

void main()
{
     ::ifstream infile("C:\\abc.txt");

     int flag=0;
     char cVar;
     string word,line;
     int iLineNum=1;
     int iWordNum=1;
```

```
typedef pair<int,int> location;
location loc;
typedef vector<location> lvec;
lvec temp;
vector<location>::iterator liter;

map<string,lvec> wordmap;
map<string,lvec>::iterator iter;

map<int,string> linemap;

while(infile)
{
     infile.get(cVar);

     if(cVar==' ' || cVar=='.' || cVar==',' ||
        cVar==';' || cVar=='\n')
     {
          if(flag==0)
          {
               loc.first=iLineNum;
               loc.second=iWordNum;
               iWordNum++;
               iter=wordmap.find(word);
               if(iter!=wordmap.end())
                    (iter->second).push_back(loc);
               else
               {
                    temp.push_back(loc);
                    wordmap[word]=temp;

                    temp.erase(temp.begin(),
                              temp.end());
               }
               word.erase(); //nullifying the string
               flag=1;
          }

          if(cVar!='\n')
               line=line+cVar; //adding a character to
                               //the string
          else
          {
               linemap[iLineNum]=line;
               line.erase(); //nullifying the string
               iLineNum++;
               iWordNum=1;
          }
          continue;
     }
     else
```

```
                flag=0;
            word=word+cVar; //adding a character to the
                            //string
            line=line+cVar;
        }
        while(1)
        {
             cout<<"Please enter the word to be searched "
                 <<(enter blank to quit): ";
             word.erase();
             while(cin)
             {
                  cin.get(cVar);
                  if(cVar=='\n')
                       break;
                  word=word+cVar;
             }
             if(word.empty()) //if string is empty
                  break;
             iter=wordmap.find(word);
             //string::c_str()returns the contained string
             cout<<"\nNumber of occurrences of '"
                 <<word.c_str()<<"' = "
                 <<iter->second.size()<<endl<<endl;
             for(liter=iter->second.begin();
                 liter!=iter->second.end();liter++)
                  cout<<"("<<liter->first<<","
                      <<liter->second<<") "
                      <<linemap[liter->first].c_str();
             cout<<endl<<endl;
        }
        cout<<"\nBye!\n\n";

    }
    /*End of textQuerySearch.cpp*/
```

Explanation of the Code

The code can be broadly divided in two steps as follows:

Step 1: Create a map of words with their locations.

```
64 (5,6)
A (2,1) (12,11)
Chess (1,1) (7,6)
India (8,6)
Invented (8,4)
It (3,2)
Of (9,6)
```

```
The (11,10)
To (10,5)
What (4,4)
a (2,6) (2,11) (3,4) (4,8) (6,2) (7,8) (8,8) (10,7)
ability (14,4)
alike (12,10)
all (9,2) (9,7)
an (14,2)
and (2,10) (4,2) (5,2) (7,3) (8,1) (11,8) (12,7) (14,1)
appears (4,5)
as (8,7)
be (4,7) (10,6)
been (12,3)
better (13,8)
bishops (4,1)
black (5,1)
board (5,4)
brilliant (7,4)
by (12,5) (12,8)
chess (2,4) (5,11)
clash (2,12)
commoners (12,9)
concentration (13,9)
cunning (7,1)
deduce (14,6)
dimensional (4,10)
facts (14,7)
followers (9,1)
for (5,9)
from (14,8)
game (2,2) (3,5) (8,10) (11,1) (12,1) (13,5)
hard (11,5)
has (7,7) (8,12) (9,11) (12,2)
history (8,3)
improved (14,3)
in (8,5)
intellectual (1,5)
intelligence (2,9)
is (1,2) (2,5) (3,3) (5,8)
it (8,11) (9,10)
kings (3,7) (12,6)
knights (3,10)
known (1,8)
labor (11,6)
largest (10,3)
leads (13,6)
literature (10,4)
logic (14,9)
long (8,2)
mankind (1,10)
```

```
master (6,1) (10,9)
mind (1,6)
most (1,4)
multicolored (6,4)
multidimensional (6,3)
of (2,3) (2,8) (2,13) (3,6) (5,5) (6,6) (10,10) (11,4)
(13,3)
over (9,3)
pawns (4,3)
perhaps (10,1)
played (12,4)
practice (11,9) (13,2)
queens (3,8)
regular (13,1)
requires (11,2)
rich (7,9)
rooks (3,9)
sport (1,7)
sports (9,9)
squares (5,7)
strategy (7,2)
study (11,7)
tactics (7,5)
the (1,3) (5,10) (9,4) (9,8) (10,2) (10,11) (13,4)
to (1,9) (4,6) (13,7) (14,5)
true (10,8)
two (4,9)
war (2,7) (8,9)
white (5,3)
wills (3,1)
wonderland (6,5)
world (9,5)
years (11,3)
```

Step 2: Create a map of lines.

```
1 Chess is the most intellectual mind sport known to
mankind.

2 A game of chess is a war of intelligence and a clash of

3 wills. It is a game of kings, queens, rooks, knights,

4 bishops and pawns. What appears to be a two dimensional

5 black and white board of 64 squares is, for the chess

6 master, a multidimensional multicolored wonderland of

7 cunning strategy and brilliant tactics. Chess has a rich

8 and long history. Invented in India as a war game, it has
```

```
9 followers all over the world. Of all the sports, it has

10 perhaps the largest literature. To be a true master of
the

11 game requires years of hard labor, study and practice.
The

12 game has been played by kings and by commoners alike. A

13 regular practice of the game leads to better
concentration

14 and an improved ability to deduce facts from logic.
```

The detailed explanation

Let us go straight to the `while` loop. The loop reads the characters from the file one by one. If the first character is neither a punctuation mark nor the new line character, we simply add it to the string representing a word (second last line of the `while` loop).

```
word=word+cVar;
```

When a punctuation mark or the end of line is encountered,

```
if(cVar==' ' || cVar=='.' || cVar==',' || cVar==';' ||
   cVar=='\n')
```

we reckon that we have *finished* loading a word. We populate an object 'loc' with the line number and word number of the word.

```
loc.first=iLineNum;
loc.second=iWordNum;
```

We also increment 'iWordNum' because the position of the next word would be one greater than the previous one.

For the time being, ignore the test

```
if(flag==0)
```

Since our word map (see *Step 1* above) should keep a vector of all positions of each word, we must first find whether the word already exists in our word map or not.

```
iter=wordmap.find(word);
```

This statement returns an iterator. If the word *is* found in any of the first members of the word map, it points at that element whose first member is the word itself. If the word is *not* found in any of the first members of the word map, the iterator points past its end.

If the word is found in any of the first members of the word map,

```
if(iter!=wordmap.end())
```

we append the location object 'loc', which we have already populated, into the location vector, which is the second member of the element pointed at by the iterator.

```
(iter->second).push_back(loc);
```

If the word is not found in any of the first members of the word map, we populate a temporary vector of locations with only one element—the location object 'loc'.

```
temp.push_back(loc);
```

Next, we insert the word and its location vector into the word map.

```
wordmap[word]=temp;
```

We ensure that the temporary vector of locations remains vacant by writing the following line of code.

```
temp.erase(temp.begin(),temp.end());
```

Next, we discard the contents of the string object that is holding the just read word so that the next word can be loaded from the file.

```
word.erase();
```

The test

```
if(flag==0)
```

ensures that if more than one punctuation mark or new line character are encountered one after another, then all of them are ignored while building the word map.

On the first occasion, this test returns true. Therefore, the location of the loaded word updates the word map. The value of 'flag' has been set to '1' within this if block. This ensures that this test returns false if the next character of the text file is also a punctuation mark or the new line character since the value of the 'flag' is reset to '0' only if the character encountered is neither a punctuation mark nor the new line character

After loading the word, if the end of line character is encountered, we increment the value of line number and reset the value of word number to '1'.

```
if(cVar!='\n')
     line=line+cVar;
else
{
     . . . .
     . . . .
     iLineNum++;
     iWordNum=1;
}
```

Also, we would like to straight away read the next character from the file without any further processing. Therefore, the `continue` keyword has been used.

This finishes the explanation of how the word map has been created. Let us now focus our attention on the creation of the line map.

As we read the characters from the file, we append them into the string that represents a line.

```
line=line+cVar;
```

If the read character is a punctuation mark but not the new line character, we simply continue to append it to the line string object line (punctuation marks are a part of the line).

```
if(cVar==' ' || cVar=='.' || cVar==',' || cVar==';' ||
   cVar=='\n')
{
. . . .
     if(cVar!='\n')
          line=line+cVar;
```

If the read character is the new line character, we reckon that we have loaded one complete line into the line string object 'line', and therefore simply insert the line number and the contents of the line into the line map.

```
if(cVar!='\n')
     line=line+cVar;
else
{
     linemap[iLineNum]=line;
     line.erase();
     . . . .
```

We also erase the contents of the line string object so that the next line can be loaded afresh.

Now, we come to the last portion of the code wherein the loop accepts the word to be searched from the user and returns the results.

After prompting the user, we first clean up the variable in which the word entered by the user would be stored. The read characters are appended to the word variable till the user presses the enter key.

The test that breaks the potentially infinite loop is as follows:

```
if(word.empty())
```

It has been inserted in the middle since we don't want the rest of the loop to execute. If the user enters a blank string, the loop breaks and the program terminates.

If the user does not enter a blank string, we try to find it in the word map.

```
iter=wordmap.find(word);
```

If the word is found in the word map, this iterator points at the element whose first member is the word itself and whose second element is a vector of locations of the found word. The size of this vector gives us the number of occurrences of the found word.

The `for` loop

```
for(witer=iter->second.begin();
    witer!=iter->second.end();witer++)
```

iterates through the vector of locations of the found word. Each element of the vector is a pair of line position and word position. The locations are displayed by enclosing these positions in brackets and separating them by commas.

```
cout<<"("<<witer->first<<","<<witer->second<<") ". . .
```

Passing the line position to the line vector returns the corresponding line. This is also displayed in the `for` loop.

```
cout<<. . .<<linemap[witer->first].c_str()<<endl;
```

Comparison of C++ with C

C++ is an extension of C language. It is a proper superset of C language. This means that a C++ compiler can compile programs written in C language. But, the reverse is not true. A C++ compiler can understand *all* of the keywords that a C compiler can understand. Again, the reverse is not true. Decision making constructs, looping constructs, structures, functions etc. are written in *exactly* the same way in C++ as they are in C language. Apart from the keywords that implement these common programming constructs, C++ provides a number of additional keywords and language constructs that enable it to implement the Object-oriented paradigm.

Differences between C++ and C can be divided into two categories:

- Non-object-oriented features provided in C++ that are absent in C language.
- Object-oriented features provided in C++ to make it comply with the requirements of the Object-Oriented Programming System.

Non-object-oriented Features Provided in C++ that are Absent in C Language

Enumerated data types

An enumerated data type in C is internally treated as an integer. In C++, it is treated as a separate data type in its own right. Direct conversion from an integer to the enumerated data type is therefore prohibited.

```
enum day_of_week
{
  monday,
  tuesday,
  wednesday,
  thursday,
  friday,
  saturday,
  sunday
};

day_of_week d;
d=Monday; //OK
d=2;      //ERROR
```

Reference Variables

(Refer to Chapter 1 for a detailed discussion.)

Constants

In C, it is illegal to use an integer, which has been declared as a constant, to specify the size of an array. This is not so in C++.

```
const int size=100;
char cArr[size]; //legal in C++ but illegal in C
```

Function prototyping

(Refer to Chapter 1 for a detailed discussion.)

Function overloading

(Refer to Chapter 1 for a detailed discussion.)

Functions with no default values for arguments

(Refer to Chapter 1 for a detailed discussion.)

Functions with no formal arguments

In C, it is reckoned that a function that has no formal arguments accepts an unspecified number of parameters. Therefore, it is legal to pass parameters to it.

In C++, it is reckoned that a function that has no formal arguments does not accept parameters. Therefore, it is illegal to pass parameters to it.

Inline functions

(Refer to Chapter 1 for a detailed discussion.)

Object-oriented Features Provided in C++ to make it Comply with the Requirements of the Object-Oriented Programming System

The following are some of the additional keywords that have been provided in C++ to make it an object-oriented programming language:

- `class` (Refer to Chapter 2 for a detailed discussion.)
- `friend` (Refer to Chapter 2 for a detailed discussion.)
- `operator` (Refer to Chapter 8 for a detailed discussion.)
- `private` (Refer to Chapter 2 for a detailed discussion.)
- `protected` (Refer to Chapter 5 for a detailed discussion.)
- `public` (Refer to Chapter 2 for a detailed discussion.)
- `template` (Refer to Chapter 9 for a detailed discussion.)
- `this` (Refer to Chapter 2 for a detailed discussion.)
- `virtual` (Refer to Chapter 6 for a detailed discussion.)

Appendix C

Comparison of C++ with Java

C.1 Similarities between C++ and Java

The following are some of the features that make C++ and Java similar:

- **Comments**

  Comments are given in Java programs in exactly the same way as they are given in C++ programs. The multiline comments (`/* ... */`) and single-line comments (//) of C++ are supported in Java also.

- **Control structures**

  Decision-making and looping constructs of C and C++ are provided in Java also. Moreover, exactly the same syntax is used for utilizing them in Java source codes.

- **Keywords for implementing exception handling**

  The keywords `try`, `catch`, and `throw` that are provided in C++ are provided in Java. Moreover, the same syntax is used for utilizing them in Java source codes. However, there is a slight difference between the way C++ allows all unhandled exceptions and the way Java does. This is explained in the section on 'Differences between C++ and Java'.

- **Fundamental data types**

  Like C++, Java also provides a set of fundamental data types. They are:

Type	*Size*
`byte`	1 byte (signed 8-bit)
`boolean`	1 byte (signed 8-bit)
`char`	2 bytes Unicode (signed 16-bit Unicode)
`short`	2 bytes (signed 16-bit)
`int`	4 bytes (signed 32-bit)
`long`	8 bytes (signed 64-bit)
`float`	4 bytes (signed 32-bit)
`double`	8 bytes (signed 64-bit)

- **Declaration of objects**

  Variables of primitive types are declared in exactly the same way in Java as in C++. The following statement declares an integer type variable in both C++ and Java.

  ```
  int x;
  ```

  However, it is different in the case of classes. The section on 'Differences between C++ and Java' explains this difference.

- **Purpose of the class construct**

  Like C++, Java also provides the class `construct`. The purpose and functionality of this construct is approximately the same in both languages. Like classes that are defined in C++, there are member functions and member data in the classes that are defined in Java. However, there are differences between the ways this construct has been implemented in the two languages. The section on 'Differences between C++ and Java' explains these differences.

- **The static keyword**

  The purpose of this keyword is the same in both the languages.

- **Constructor**

  Constructors in Java are defined in exactly the same way as they are defined in C++ and also serve the same purpose.

  Constructors in C++ and Java are similar to each other in the following ways: The compiler defines the constructor if we do not define one. If we define the zero-argument constructor or a parameterized constructor for a class, the compiler does not define the default constructor for the class. An access specifier can be specified to a constructor.

- **Inheritance**

  Like C++, Java also supports inheritance. However, Java does not support multiple inheritance. This is explained in the section on 'Differences between C++ and Java'.

- **Static polymorphism**

  Like C++, Java supports static polymorphism. Two functions with different signatures can have the same name.

- **Dynamic polymorphism**

  Dynamic polymorphism is supported in both C++ and Java. If a member function of the base class has been overridden in the derived class and it is called for a base class reference that actually refers to a derived class object, then the member function of the derived class gets called.

- **Overriding member functions of a base class in its derived classes**

  Base class member functions can be overridden in the derived class except when the base class function has been specified as final. This exceptional case has been explained in the section on 'Differences between C++ and Java'.

C.2 Differences between C++ and Java

The following are some of the features that make C++ and Java different:

- **Structures and unions**

  There are no structures and unions in Java. Java supports only classes.

- **The main() function**

  Unlike the 'main()' function in C++, the 'main()' function in Java is not a global function. It is instead a public static function of a class. The class that has such a 'main()' function is executed by clients or from command line.

  Unlike the 'main()' function in C++, which takes an array of character pointers as parameter, the 'main()' function in Java takes an array of objects of the class 'String' as parameter.

- **Header files versus packages**

  Instead of header files, we have packages in Java. Packages in Java serve a similar purpose as header files in C++. Packages are included in Java source codes by using the import directive as follows:

  ```
  import java.util.Date;    //importing a package
  ```

  This statement imports the 'Date' class, which is defined in the 'util' package, which is in turn included in the 'Java' package.

- **The zero-fill right shift operator (>>>)**

  Java introduces a new right shift operator—the zero-fill right shift operator. The normal right shift operator (>>), fills up the bits on the left with the value of the first bit as it shifts the bits to the right. In contrast, the zero-fill right shift operator (>>>), fills up the bits on the left with zeros as it shifts the bits to the right. This operator can be used as follows:

```
x=x>>>1;     //shifts bits in x to the right by one place
             //and move a zero from the left
```

There is also a complimentary zero-fill right shift assignment operator (>>>=). The foregoing statement can be rewritten as

```
x>>>=1; //shifts bits in x to the right by one place and
        //move a zero from the left
```

- **Operator overloading**

  Unlike C++, operator overloading is not supported by Java.

- **External functions**

  There are no global functions in Java. All functions must be members of some class or the other.

- **The final keyword**

  The `final` keyword serves the same purpose as the `const` keyword of C++. Member variables are declared as constants by prefixing this keyword to their declarations.

- **Declaration of objects and the new operator**

  We already know that variables of primitive types are declared in exactly the same way in Java as in C++. However, the case is different in the case of classes.

  Suppose **A** is a class. The following statement creates an actual object in C++.

```
A A1;    //A1 is an object in C++ but a null reference in
         //Java
```

But in Java, the above statement would only declare a null reference. Such a reference has to be explicitly initialized by using the `new` operator as follows:

```
A1 = new A();
```

In C++, the `new` operator captures a memory block in the heap and returns a pointer to it. In Java, the `new` operator captures a memory block in the heap and returns a reference to it. This is the only way of declaring an object in Java whereas in C++, an object may be declared either in the stack or in the heap by using the `new` operator.

- **Pointers**

  Java does not support pointers. In Java, apart from the variables of primitive data types, all objects are actually references.

  The following listing makes it evident that variables of primitive data types are always passed by value.

```
class first
{
  public static void main(String args[])
  {
    int x;                     //x is of primitive type
    x=100;

    int y;
    y=x;                       //y is a separate memory
                               //location

    //outputting to the console
    System.out.println("Before changing:");
    System.out.println("x=" + x + ",y=" + y);

    x=200;                     //x changed, y unchanged

    System.out.println("After changing:");
    System.out.println("x=" + x + ",y=" + y);
  }
}
```

Output

```
Before changing:
x=100,y=100
After changing:
x=200,y=100
```

Listing C.1 Variables of primitive data types are passed by value

Class objects are always references. The following listing illustrates this.

```
class A
{
  public int x;   //public member
}

class first
{
  public static void main(String args[])
  {
    A A1 = new A();      //necessary to initialize since A1
                         //is only a reference ... now A1 is
                         //a reference to a memory location
    A1.x=100;

    A A2;
    A2=A1;               //A2 is also a reference to the
                         //memory location to which A1 is a
                         //reference

    //outputting to the console
    System.out.println("Before changing:");
    System.out.println("A1.x=" + A1.x + ",A2.x=" + A2.x);

    A1.x=200;            //A1.x changed, A2.x also changed

    System.out.println("After changing:");
    System.out.println("A1.x=" + A1.x + ",A2.x=" + A2.x);
  }
}
```

Output

```
Before changing:
A1.x=100,A2.x=100
After changing:
A1.x=200,A2.x=200
```

Listing C.2 Class objects are passed by reference

- **Garbage collection**

Unlike C++, where dynamically acquired memory must be explicitly returned to the OS, garbage collection is automatic in Java. During the execution of a Java program, an automatic garbage collector runs in the background. If it finds any locked up memory that is no longer being referenced, it returns it to the OS.

- **Destructor**

  There is no `delete` keyword in Java. Therefore, there are no destructors in Java.

  In Java, the programmer can simply create an object by using the `new` operator. There is no need to worry about reclaiming the memory, that is the garbage collector's responsibility.

  The 'finalize()' method in Java approximates the destructor's behavior. However, there is a difference. The garbage collector will definitely execute this method for an object that it is destroying. But the exact instance at which the garbage collector would destroy an object cannot be specified.

- **Terminating class definitions, access specifiers, defining member functions**

  Class definitions are terminated by semicolon in C++. In Java, class definitions are not terminated by a semicolon.

  In C++, access specifiers are specified for a group of class members. In Java, access specifiers are specified for each individual class member separately.

  Class member functions may be defined outside the class in C++. In Java, class member functions are always defined inside the class.

  Unlike C++, access specifiers can be specified for classes in Java. A class prefixed with the keyword `public` is visible to classes outside the package in which it was created. Otherwise, it is visible to classes of the same package only.

  Consider the following class written in C++:

```
class A
{
  private:          //access specifiers provided to a group
                    //of members in C++
    int x;
  public:
    void setx(int);
    int getx();
};

public class A
{
  private int x;  //access specifiers provided to
                  //individual members in Java
  public void setx(int p)
  {
    x=p;
  }
```

```
    public int getx()
    {
      return x;
    }
  }
```

Listing C.3 Access specifiers provided to individual members in Java

Apart from `private`, `protected`, and `public` access specifiers, which are also provided by C++, Java provides the `package` access specifier. The `package` access specifier makes a member or a class to which it is applied visible to other classes of the same package only. It is the default access specifier. The functionality of this access specifier is similar to that of namespaces in C++.

- **Enumerations**

Java does not support the enum keyword. However, an enumerated type can be created in Java as a class that has only static final data members.

```
public class Color     //creating an enumerated type
{
  public static final int red=1;
  public static final int blue=2;
  public static final int green=3;
  public static final int yellow=4;
  public static final int brown=5;
}

. . . .
. . . .

if(fontColor==Color.red)    //using a value of the
                            //enumerated type
  . . . .
```

Listing C.4 Specifying and using enumerated data types in Java

In C and C++, while using the values of enumerated types, we need not qualify them by the name of the enumerated type itself. This leads to potential name clashes. This drawback does not exist in Java since the value of an enumerated type is qualified by the name of the enumerated type itself.

- **The this keyword**

  The `this` keyword provides the same functionality in Java as it does in C++ with the difference that in C++ it is a pointer whereas in Java it is a reference. Therefore the 'this' pointer need not be dereferenced in Java.

```
class A
{
  private int x;
  void setx(int x)
  {
    this.x=x;      //'this' is a reference in Java
  }
}
```

Listing C.5 'this' is a reference in Java

In the above example, the `this` pointer was used to resolve name ambiguity. The name of the member variable of class A is the same as that of the formal argument of the 'setx()' function of class A. Using the `this` keyword resolves this ambiguity and the value of the member variable gets set to that of the formal argument.

- **The syntax of inheritance**

  Java provides the `extends` keyword for declaring a derived class. This keyword can be used as follows:

```
class B extends A   //class B inherits from A
{
}
```

- **The default base class in Java**

  If you do not specify a base class for a class you are defining, the Java compiler automatically assigns a class called 'Object' as its base class. Therefore, all classes in Java inherit from the class 'Object'.

- **Overriding member functions of a base class in its derived classes**

  Using the `final` keyword in the declaration of the member function of a class prevents the derived class from overriding it. The compiler throws an error if a member function of the derived class overrides a final method of the base class.

- **Interfaces and abstract base classes**

  An interface in Java is similar to the abstract base class in C++ but with the following differences:

  - Member functions of an interface can only be declared. They cannot be defined.
  - Member data of an interface are considered final.

  An interface is declared in Java as follows:

```
public interface A
{
  public void abc();      //declaration only ... no definition
  public void def();      //declaration only ... no definition
}
```

  Like the abstract base classes of C++, interfaces in Java cannot be instantiated. The following piece of code is illegal:

```
A A1 = new A();
```

  A class that implements an interface can be instantiated provided it defines all member functions of the interface.

```
public class B implements A //syntax for implementing an
                            //interface
{
  public void abc()
  {
    /*
      definition of the function
    */
  }
}
```

Listing C.6 Declaring and implementing an interface in Java

If not all member functions of the interface are defined in the class that implements them, the class becomes an abstract base class and cannot be instantiated.

Multiple inheritance is not supported in Java. However, Java provides interfaces. A class in Java can inherit from only one class but implement an unlimited number of interfaces. In the following definition, class X inherits from class P but implements the interfaces A and B.

```
//implementing more than one interface
public class X extends P implements A, B
{
  /*
    definition of class X
  */
}
```

An abstract class is declared in C++ by declaring at least one of its member functions as a pure virtual function. An abstract class is declared in Java by using the `abstract` keyword in its declaration. As in C++, an abstract class cannot be instantiated.

```
public abstract class A      //an abstract class
{
  public abstract void abc();    //an abstract method of
                                 //an abstract class
  public void def()              //an non-abstract method
                                 //of an abstract class
  {
    /*
      definition of the function
    */
  }
}
```

Listing C.7 An abstract class in Java

As can be seen, some functions of an abstract class can be declared as abstract by using the `abstract` keyword in their declaration while the others can be declared as non-abstract by not using the `abstract` keyword in their declaration. In contrast to interfaces, member functions of an abstract class can be defined. If not all member functions of an abstract class are defined in the class that extends it, the latter class also becomes an abstract base class and cannot be instantiated.

Abstract and non-abstract member functions of the abstract base class can be overridden in the derived class.

```
abstract class A
{
  public abstract void abc();
  public void def()
  {
    System.out.println("def() function of class A");
  }
}

public class B extends A
{
  public void abc()     //overriding abstract function of
                        //base class
  {
    System.out.println("abc() function of class B");
  }
  public void def()     //overriding non-abstract function
                        //of base class
  {
    System.out.println("def() function of class B");
  }
  public static void main(String args[])
  {
    A A1 = new B();
    A1.abc();
    A1.def();

    B B1 = new B();
    B1.abc();
    B1.def();
  }
}
```

Output

```
abc() function of class B
def() function of class B
abc() function of class B
def() function of class B
```

Listing C.8 Overriding base class functions in the derived class

The foregoing output makes one thing very clear. All non-static member functions in Java are virtual functions. The abstract qualifier is used in the declaration of a member function of the base class only to force its override in the derived class.

An overridden member function of the base class can be called from the member function of the derived class by using the keyword super as follows:

```
abstract class A
{
  public abstract void abc();
  public void def()
  {
    System.out.println("def() function of class A");
  }
}

public class B extends A
{
  public void abc()
  {
    System.out.println("abc() function of class B");
  }
  public void def()
  {
    super.def(); //calling an overridden function of the
                 //base class
    System.out.println("def() function of class B");
  }
  public static void main(String args[])
  {
    A A1 = new B();
    A1.abc();
    A1.def();
  }
}
```

Output

```
abc() function of class B
def() function of class A
def() function of class B
```

Listing C.9 Calling an overridden function of the base class in a member function of the derived class

- **Base class initialization**

Base class member data are initialized in C++ by the member initialization list. In Java, using the super keyword achieves this objective.

```
class A
{
  private int x;
  public A(int p)
  {
    x=p;
  }
}

class B extends A
{
  int y;
  public B(int p, int q)
  {
    super(p);    //calling the base class constructor
    y=q;
  }
}
```

Listing C.10 Initializing bas class members from the derived class constructor

- **Exception handling**

  The exception handling mechanism provided in Java is very similar to C++ with the following differences:

  - In Java, the classes of all thrown objects must inherit from the class 'Throwable'. Therefore, the block 'catch(Throwable)' in a 'try ... catch' construct in Java is equivalent to the catch(...) block in C++.
  - Java introduces a new keyword `finally` to be used in the 'try ... catch' block. The block labeled `finally` is always executed at the end of a 'try ... catch' block.

Appendix D

Object-Oriented Analysis and Design

D.1 Introduction

This appendix gives a brief but comprehensive overview of Object-Oriented Analysis and Design (OOAD). OOAD is a design methodology. It is used to model solutions of software engineering problems. Models can be translated into actual code written in object-oriented languages.

Why Build Models?

Software engineering problems are usually quite complex. Different aspects of the solution need to be modeled using standard notations. After these models have been verified for correctness, they are implemented in actual code.

Models serve several purposes. Some of them are as follows:

- The solution can be tested for correctness and completeness before actually building it.
- Models help the developers in communicating clearly and precisely with customers and also among themselves. This ensures that all parties are in sync with each other.
- Models help developers in visualizing the solution clearly.
- Complexity of the problem gets reduced since the entire system can be broken down into successively smaller portions.

Overview of OOAD

What is OOAD?

OOAD has the following stages:

1. **Analysis:** In this phase, a model of the solution is built. The analysis model contains classes with their members, their relationships etc. The analysis model shows *what* the desired solution must do, not *how* it will be done. It does not contain any implementation details.

2. **System design:** In this phase, the analysis model is divided into manageable sub-systems. Relationships amongst these sub-systems are also modeled. A strategy of attacking the problem is formulated. Performance optimization is also finalized.
3. **Object design:** In this phase, implementation details are added to the model built during the analysis phase.
4. **Implementation:** The object model thus created is finally translated into a particular programming language.

Overall development time is not always less in OOAD as compared to the conventional methodology. But the benefit is that the resulting model is better suited for future reuse. Downstream errors and maintenance efforts also get reduced.

D.2 The Object-Oriented Model

An object-oriented model consists of the following three kinds of models:

- Object model
- Dynamic model
- Functional model

The object model describes the objects in the system and the relationships amongst these objects. It consists of object diagrams.

The dynamic model describes how objects in the system interact with each other. It consists of state diagrams. A state diagram depicts states and transitions between states that are caused by events.

The functional model describes how data gets transformed in the system. It consists of data flow diagrams. A data flow diagram depicts processes and data flow among the processes.

The three models complement each other. They are linked to each other. The object model is described first. It is necessary to describe what is changing or transforming before describing when and how it changes.

The object model describes classes upon whom the dynamic and functional models operate. The operations in the object model relate to events in the dynamic model and functions in the functional model.

The Object Model

Object diagrams are of two types: class diagrams and instance diagrams.

As its name suggests, a class diagram describes classes and relationships amongst classes.

An instance diagram depicts the relationship amongst a particular set of objects that exist together at a given instance of time. A large number of instance diagrams can be generated from a single class diagram.

Boxes are used to depict objects and classes. Different object-oriented design tools use slightly different variations of these boxes. A sample is shown below:

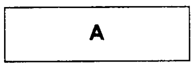


Diagram D.1 A class is depicted in an object model as a box with sharp corners; name is in bold

Attributes

Attributes are nothing but data members of classes. They are listed in the second part of the class box. Each attribute name may be followed by a colon and its type. This, in turn, may be followed by an equality symbol and the default value of the data member. A line is drawn between the class names and attributes. No such line exists in object boxes.

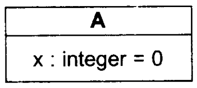


Diagram D.2 Depicting attributes in class box

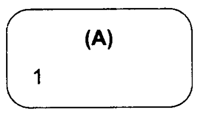


Diagram D.3 An object is depicted in an object model as a box with rounded corners; class name is in bold but is surrounded by parentheses

Operations

Operations are nothing but member functions of classes. They are listed in the third part of the class box. The name of the operation is followed by a list of formal arguments in parentheses. The arguments are mentioned in the same way as the attributes. These parentheses may be followed by a colon and the result type of the operation.

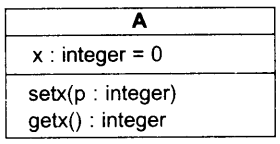


Diagram D.4 A class with operations

Links and associations

An association depicts a conceptual or physical relationship between two classes. A link is an instance of an association. Associations may be bidirectional or unidirectional. An association may be implemented as a pointer from one object to another. The notation for an association is a line between the associated classes.

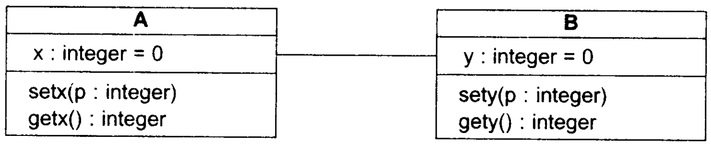


Diagram D.5 Association

Multiplicity

Multiplicity signifies the number of instances of one class that may relate to a single instance of an associated class. An association may be:

- One-to-one
- One-to-many
- Many-to-many

A ball at one end of the line that depicts association between two classes indicates a 'many' side. A hollow ball indicates that zero or one instance of the class on whose side the ball appears may be associated with an instance of the associated class. A solid ball indicates that zero or many instances of the class on whose side the ball appears may be associated with an instance of the associated class. If no balls appear, it indicates a one-to-one relationship.

If possible, the exact permissible number of instances of one class that can be associated with one instance of the associated class is also specified. An exact value can be specified. An interval of values can also be specified. The interval may be a single interval or a set of disconnected intervals.

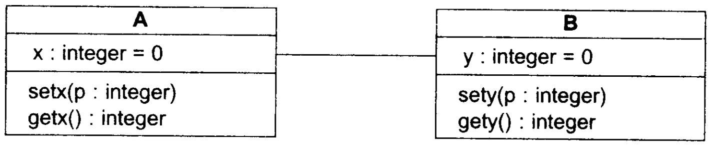


Diagram D.6 A one to-one association

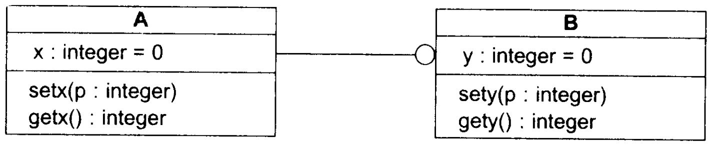


Diagram D.7 A one-to-many association (zero or one instance of class B may be associated with an instance of class A)

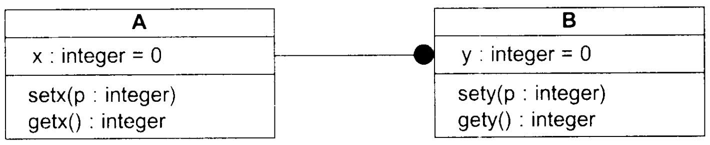


Diagram D.8 A one-to-many association (exactly two instances of class B may be associated with an instance of class A)

Some of the other ways of specifying the multiplicity in Diagram D.8 are:

- 2+ (2 or more),
- 2–4 (2,3, or 4),
- 2,5,18 (either 2 or 5 or 18), etc.

Association attributes

An association may have its own attributes. Association attributes are depicted in boxes attached to the association by a loop. Such boxes have the same characteristics as the boxes that are used to represent classes.

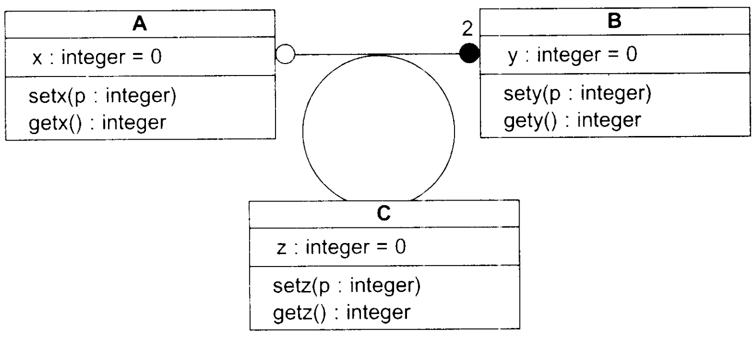


Diagram D.9 Association attributes

Attributes of a many-to-many association are always properties of the associations itself. They cannot be attached to either object. On the other hand, it is possible to insert attributes for one-to-one and one-to-many associations into the class opposite the 'one' side.

Pointers are embedded either in one or both of the associated classes to implement association. Alternatively, if a separate class has been used to implement an association as in Diagram D.9, pointers to both classes appear in the third class. All this is explained in the last section of this appendix.

Aggregation

In aggregation, an object of one class contains objects of other classes. Like association, a line connects two classes between whom an aggregation relationship exists. However, a diamond appears next to the container class.

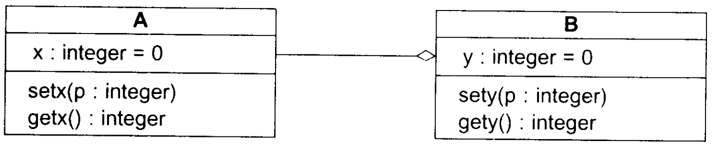


Diagram D.10 Aggregation—an object of class B contains an object of class A

Inheritance

We have already studied inheritance in Chapter 5. The notation for inheritance is an upright triangle connecting a superclass to its subclasses. The superclass is connected by a line to the apex of the triangle.

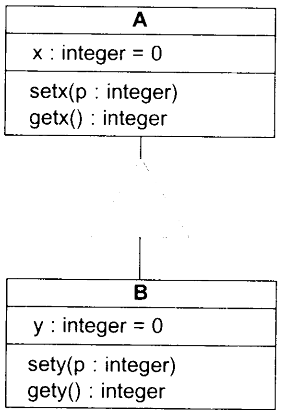


Diagram D.11 Inheritance—class B derives from class A

Abstract classes

Abstract classes have already been explained in Chapter 6. An abstract function (that makes the class abstract) is designated by a comment in braces.

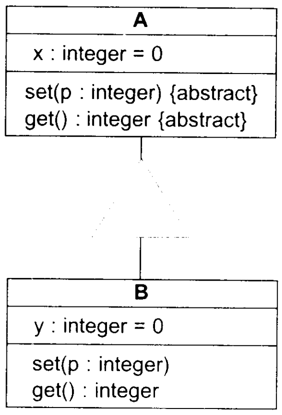


Diagram D.12 Depicting an abstract base class

Multiple inheritance

We have already studied multiple inheritance in Chapter 5. Multiple inheritance is depicted by using the same symbols that are used for single inheritance (See diagram D.11).

The Dynamic Model

The dynamic model models the sequence of changes that occur in a system.

Two important concepts in dynamic modeling are events and states. State of an object is represented by the set of values of the object at a given point of time. Events are external stimuli that cause a change of state.

A state diagram depicts the states, events and transitions from one state to another for a given class. One state diagram is created for each class that exhibits important dynamic behavior. The set of all such state diagrams constitutes the dynamic model. The state diagrams shared events with each other.

Events

An event is an instantaneous occurrence that causes a change of state. An event is a one-way transmission of information from one object to another.

Scenarios and event traces

A scenario is a textual line-by-line description of the sequence of events that occur during one particular execution of a system. Event traces are created for each scenario. For this, the sender and the receiver objects of each event are identified. This event trace shows each of these objects as a vertical line. Each event is depicted as a horizontal arrow from the sender object to the receiver object.

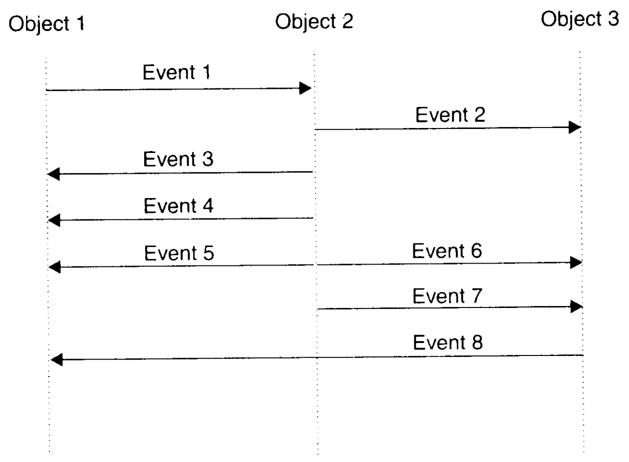


Diagram D.13 An event trace

States

A state is a set of attribute values and links of an object that can be grouped together because they occur together at a given point of time. An object remains in the same state during the interval between two events.

State diagrams

A state diagram depicts the relation between events and states. Upon receiving an event, the state the recipient object attains depends on its current state as well as the event. Transition is the change of state caused by an event.

The symbol for a state is a rounded box that may contain a name for the state. A transition is depicted as an arrow from the receiving state to the target state.

A sequence of events on the event trace corresponds to a path through the state diagram.

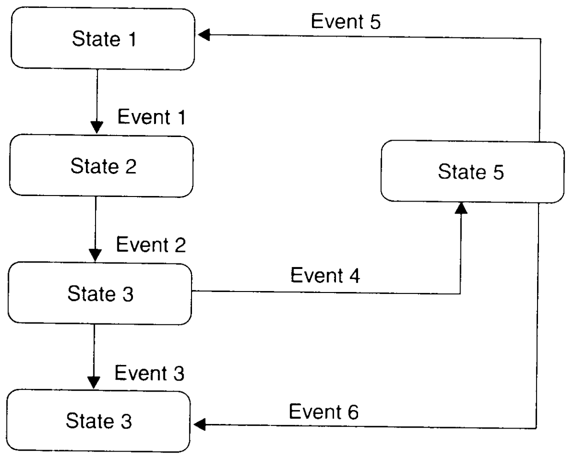


Diagram D.14 A state diagram

Conditions

A condition may influence the firing of a transition. Such a transition is known as a guarded transition. It fires when its event occurs, but only if the guard condition is true. A guard condition on a transition is shown in brackets following the event name.

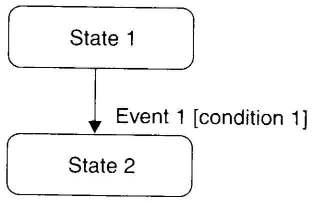


Diagram D.15 Guarded transitions

Activity

An activity is an operation that executes over a period of time. An activity may continue to execute as long as an object remains in a particular state. Activities may run continuously or terminate on their own after an interval of time. The notation for specifying that an activity A executes as long as the state remains is "*do: A*" within the state box.

Actions

An action is an instantaneous occurrence that occurs when an event occurs. An action on a transition is denoted by a slash ('/') and its name, following the name of the event that causes it.

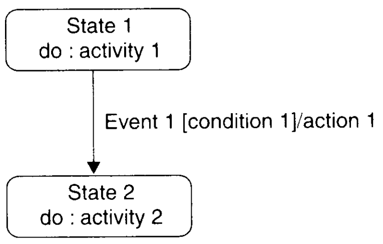


Diagram D.16 Activities and actions

Relation of object and dynamic models

The dynamic model specifies the sequences in which operations of an object can be called. States represent the attribute and link values for the object that may exist concurrently. Events are nothing but operations on the object.

The Functional Model

The functional model depicts the transformations of input values into output values. The order in which the transformation takes place is not modeled here. Transformations are depicted using data flow diagrams. The functional model consists of multiple data flow diagrams.

Data flow diagrams

A data flow diagram depicts the flow of data values from source objects through processes to their destinations objects.

A data flow diagram contains processes and data flows. Processes transform data. Data flows depict the flow of data amongst processes, actor objects and data store objects. Actor objects are passive objects that enter data and use the data produced by the system. Data store objects merely store data and do not transform the data in any way.

Processes

A process is depicted as an ellipse in the data flow diagram. A descriptive name of the transformation appears as its label. The input and output data for each process are also depicted.

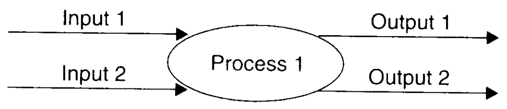


Diagram D.17 A process in a data flow diagram

Data flows

A data flow connects the output of one object or process to the input of another object or process. The value is not changed by the data flow.

Actors

An actor initiates the data flow by inputting values. Values output by a data flow diagram may also terminate with an actor. Actors may be data-entry operators or timed devices. An actor is depicted as a rectangle since an actor is inherently an object.

Data stores

A data store merely stores data for later use. It does not have any operations defined. It is depicted as a pair of parallel lines with its name in between the lines. Data stores are inherently objects.

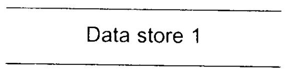


Diagram D.18 A data store

A data flow diagram

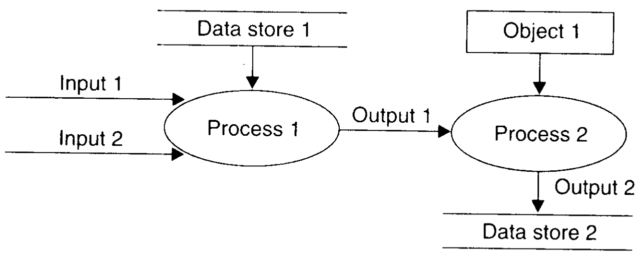


Diagram D.19 A data flow diagram with data flows, processes, actors and data stores

D.3 Analysis

Now we know the three models that constitute an object-oriented model. This section will teach us how we can create a model from the requirement document. The last section will teach us how to implement the model.

Overview of Analysis

Requests generated by users, developers, and managers are first collected and consolidated. A problem statement is created from this consolidated list. Models are built by analyzing the problem statement and by taking inputs from users, from domain experts, and from the developer's knowledge of the real world.

Large models are built up iteratively. A small model to achieve only the core requirement is built first. Once the model has been perfected, it is expanded to add all ancillary functionalities.

Object Modeling

The steps for object modeling are as follows:

1. Identify classes
2. Identify associations and aggregations
3. Add attributes to classes and associations
4. Combine classes using inheritance
5. Add operations to classes *after* constructing the dynamic and functional models

Classes often appear as nouns in the problem statement.

Associations often appear as verbs or verb phrases.

Attributes often appear as nouns followed by possessive phrases.

Identify classes that share common features and designate them as base classes. Look for nouns that appear with different adjectives. The common noun can usually be modeled as a base class whereas each of the nouns with an adjective can be modeled as its derived class.

Dynamic Modeling

Briefly, the first step in dynamic modeling is to identify events. Events appear as external stimuli and responses. The next step is to summarize permissible event sequences for each object with a separate state diagram.

We can start with creating scenarios of typical executions. All common interactions should get depicted by scenarios. The next step is to create event traces for each of these scenarios. Scan the columns in the event trace to identify events that occur on each object.

States that an object attains can be identified from the interval between all pairs of events that occur on it one after the other. These events and states are organized in a state diagram. The resulting set of state diagrams constitutes the dynamic model.

This process can be repeated incrementally for special interactions such as omitted inputs, violation of domain constraints, etc. Finally scenarios for error cases are prepared. These scenarios are also merged into the state diagram by attaching the new event sequence to the existing state as an alternative path.

Functional Modeling

Activities or actions in the state diagrams of classes can be modeled as processes on the data flow diagram. Objects and attribute values of an object diagram can be modeled as flows on a data flow diagram.

Parameters of events from the dynamic model appear as input and output values of processes. The data flow diagram can be constructed by inserting processes between corresponding input and output values.

Processes in the top-level data flow diagram may be quite complex. Create a simplified and detailed lower-level data flow diagram for each such process. If this level still contains complicated processes, repeat the process till you reach a data flow diagram with very simple processes.

Finally, write a textual description of each lowest-level process. The description should emphasize the objective of the process and not how the process would get implemented.

Identify and model actors and internal storage also.

D.4 System Design

System design is the high-level strategy for solving a problem and building a solution. The system is organized into sub-systems. Sub-systems are allocated to hardware and software components. Major conceptual and policy decisions that form the guidelines for the detail design are also taken. The overall organization of the system is called the system architecture. During system design, overall high-level implementation decisions are made. This is followed by similar decisions for successively lower levels.

Breaking the System into Sub-systems

A system is divided into sub-systems based on the similarity of *services*. A service is a set of functions that have a common objective.

Each sub-system provides a well-defined well-abstracted interface. The interface data can be exchanged with the sub-system. It does not specify how the sub-system is implemented internally. Thus, the internal design of each sub-system can be created and modified while other sub-systems that interact with it remain unchanged.

Layers

A system can be divided into sub-systems and these sub-systems can be horizontally layered one on top of another. Each layer is built to provide services to the ones above it. Thus each layer provides the basis of designing the one above it. Layers are thus strongly coupled.

Partitions

Sub-systems can be vertically placed next to each other as partitions also. Partitions are independent or weakly-coupled sub-systems.

A combination of layers and partitions can be used to divide a system.

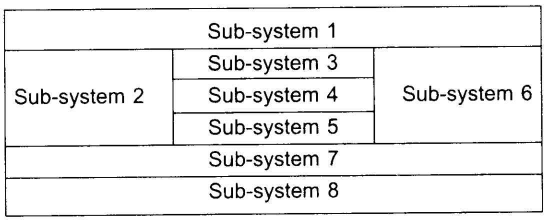


Diagram D.20 Dividing a system into sub-systems

We should also identify which activities may execute concurrently and which are mutually exclusive. The latter set of activities can be put together in a single thread of control.

D.5 Object Design

Definitions of the classes and associations modeled earlier are completed in the object design phase. Algorithms to be used in the operations are also finalized.

Overview of Object Design

The list of objects found out during analysis is revised with an intention to minimize execution time and memory.

The object model describes the classes of objects in the system, including their attributes and the operations they support.

The functional model describes the operations that the system must implement. During design we must decide how each operation must be implemented.

The dynamic model describes how the system responds to external events. The control structure for a program is primarily derived from the dynamic model.

Actions and activities of the dynamic model and the processes of the functional model are converted to operations.

While designing classes we must decide when to use values of fundamental types for data members and when to use another object. Classes can contain objects of other classes, but eventually everything must be implemented in terms of built-in primitive data types.

D.6 Implementation

Writing code is an extension of the design process. Writing code should be straightforward, almost mechanical, because all the difficult decisions should already have been made during design. The code should be a simple translation of the design into code written in a particular programming language.

Classes are declared first. Attributes and operations mentioned in the object diagram are declared as private, protected, or public members.

We have already studied how a class can be defined in C++, how members can be declared in these classes, how objects of these classes can be declared and how member functions can be called with respect to the declared objects. We have also studied how inheritance can be implemented in C++. Now is the time to carry out these activities with respect to the object design just created.

There are two general approaches for implementing associations—buried pointers and distinct association objects.

A binary association is frequently implemented as a buried pointer in each of the associated objects that point at the related object or the set of related objects of the associated class. Updating one pointer in the implementation of an association implies that the other pointer must be updated as well to keep the implementation consistent.

Association can also be implemented as an associative container class (maps and sets)

Glossary

Abstract Base Class A class that has at least one pure virtual function. Abstract base classes cannot be instantiated. They only serve as base classes for other classes to derive from. (see also pure virtual functions)

Arrow Operator (->) An operator to access data members of an object or call member functions through a pointer that points at the object.

Base Class A class from which another class inherits. (see also inheritance)

Call by Reference The formal argument of the called function is of reference type. (see also reference variables)

Call by Value The formal argument of the called function is of non-reference type. (see also reference variables)

catch The keyword used to catch a thrown exception. (see also exception handling, try, throw)

Child Class A class that inherits from another class. (see also inheritance)

Class A language construct provided by C++ that enables the implementation of object-oriented concepts like data security, guaranteed initialization of data, data abstraction, data hiding, etc. A class has member functions and member data.

Class Template A template for a class definition. Data type of data members is undefined.

Clone Functions A function that creates an object of the same type at which a base class pointer points and returns its address.

const_cast Operator This operator is used to cast away the constness of a value.

Constant Member Functions A member function that can only read the value of data members but not modify them.

Constructor A class member function that gets called automatically with respect to each object at the time of its creation. It is used to guarantee the initialization of data members of the object. (see also Guaranteed Initialization of Data)

Copy Constructor A constructor that gets called whenever an object is created and simultaneously equated to another existing object. It is called with respect to the object that is getting created while the existing object is passed as a parameter to it.

Data Member A class member that would contain the values of class objects. Each object of the class has its own copy of the data member.

Data Security An object-oriented feature that refers to preventing unauthorized functions from accessing data.

Default Constructor The constructor that gets defined by default by the compiler if we do not. It does not take parameters. (see also Zero-argument Constructor)

Default Values for Function Arguments Default values can be assigned for function arguments. These values are assigned to the arguments if no values are passed to them when the function is called.

delete Operator An operator that allows us to return a dynamically allocated block of memory to the operating system. (see also dynamic memory deallocation)

Derived Class (see child class)

Destructor A function that gets called automatically for each object at the time of its destruction. It is used to release resources held by objects.

Dot Operator (.) An operator that allows us to access data members of an object or call member functions with respect to an object.

dynamic_cast Operator A cast operator used for downcasting a pointer of base class type to a pointer of a particular derived class. If the base class pointer being typecast actually points at an object of the target type, the dynamic_cast operator returns the address of the object pointed at, else it returns NULL. If the dynamic_cast operator is used with references instead of pointers, it returns the reference to the target object or throws an exception of the type Bad_cast.

Dynamic Binding In dynamic binding, if an overridden function of the base class is called with respect to a pointer or a reference of the base class type, then which of the functions (base class function or one of the derived class versions) will actually be called, is decided based on the type of the object pointed at or referred to at run time The same function has more than one form (in the base class and the derived classes). Its call can lead to the execution of a particular version depending upon circumstances arising during run time.

Dynamic Memory Allocation In this type of memory allocation, more memory is allocated in response to requirements arising during run time.

Dynamic Memory Deallocation Once it is not required, memory allocated dynamically can be retuned to the operating system. This is dynamic memory deallocation.

Dynamic Polymorphism (see dynamic binding)

Early Binding In early binding which version of a called function that has multiple forms will be called at run time is decided during compile time itself.

Enclosing Class A class that contains the definition of another class is known as an enclosing class. (see also nested class)

Exception Handling Exception handling is a facility that enables the library code to notify error conditions, which it is incapable of handling, to the calling client. The client can catch the exception and decide upon a suitable error-handling strategy. (see also catch, throw, try)

Friend Function A non-member function that has been granted special rights to access private data members of the class of which it has been declared a friend.

Friend Class A class whose entire set of member functions has been granted special rights to access private data members of the class of which it has been declared a friend.

Function Member A function that is a member of a class. It is declared within the class. It has the right to access the private data members of all objects of the class.

Function Overloading A facility that enables the programmer to create two functions with the same name.

Function Prototype A function declaration that tells the compiler the return type of the function and the type and number of its formal arguments.

Function Template A template for a function definition. The type of its formal arguments is undefined. An actual definition of the function gets generated only when the function is called. The types of the corresponding passed parameters replace the undefined data types of formal arguments.

Generic Class (same as class template)

Guaranteed Initialization of Data The data members of a structure variable in C language may attain invalid values at its time of creation. C++ enables programmers to guarantee initialization of data members of objects by defining constructors. (see also constructor)

Inheritance A feature provided by all object-oriented languages that enables a class to inherit the data and function members of an existing class. (see also base class, parent class, super class, derived class, child class, sub class)

Inline Function A function that is expanded inline with its call by the compiler.

Late Binding (see dynamic binding)

Manipulators Manipulators are operators that enable the C++ programmer to format the output from their programs.

Multiple Inheritance A type of inheritance where a class inherits the features of more than one base class.

Mutable Data Members Data members that are never constant. Even constant member functions can modify their values. (see also constant member functions)

Namespace A language construct in C++ that allows us to divide the source code into logical parts. This helps in preventing clashes of names. Two classes with the same name can be defined if they belong to different namespace.

Nested Class A class that is defined within another class is known as a nested class. (see also enclosing class)

New Handler Function A function that gets called whenever an out-of-memory condition is encountered. We can define our own new handler function.

new Operator An operator that enables us to capture more memory in response to conditions arising during run time (see also dynamic memory allocation)

New Style Casts C++ provides the following four new style cast operators to replace the use of the old error-prone and difficult-to-detect C style casts:

- dynamic_cast
- static_cast
- reinterpret_cast
- const_cast

Object An instance of a class is known as an object.

Object-Oriented Programming System A programming system that enables programmers to model real-world objects. Data and procedures that work upon the data are bound together in a single construct called class.

Operator Overloading A feature of most object-oriented languages that enables programmers to provide additional definitions to operators so that they can take class objects as operands.

Parameterized Class (see Class Template)

Parameterized Constructor A constructor that takes parameters.

Parent Class (see Base Class)

Polymorphism A feature of object-oriented languages that allows programmers to create two functions with identical names. Polymorphism is of two types—static and dynamic. (see also Static Binding, Static Polymorphism, Early Binding, Dynamic Binding, Dynamic Polymorphism, Late Binding)

Private Class Members Function and data members of a class that have been declared under the private section of a class. Private class members can be accessed by member functions of the same class only.

Procedure-Oriented Programming System In this system, code is divided into procedures. Data and procedures that work upon the data are not bound together.

Protected Class Members Function and data members of a class that have been declared under the protected section of a class. Protected class members can be accessed by member functions of the same class and those of the derived class only.

Public Class Members Function and data members of a class that have been declared under the public section of a class. Public class members can be accessed by not only the member functions of the same class and those of the derived class. They can be accessed by global non-member functions also.

Pure Virtual Functions A special type of virtual function whose declaration is suffixed by '=0'. Presence of a pure virtual function makes its class an abstract base class. A derived class that does not override the pure virtual functions of the base becomes an abstract base class. (see abstract base class, virtual functions)

Reference Variables A reference variable is a reference to another variable. It does not occupy its own memory. It shares the memory occupied by the variable of which it is a reference.

reinterpret_cast Operator A new style cast operator that allows the conversion of one type to another.

Scope Resolution Operator An operator that enables the C++ programmer to define a member function outside the class.

static_cast Operator The only difference between the static_cast operator and the dynamic_cast operator is that while the dynamic_cast operator carries out a run-time check to ensure a valid conversion, the static_cast operator caries out no such check.

Static Binding (see early binding)

Static Data Member Static data members hold global data that is common to all objects of the class. Static data members are members of the class and not of any object of the class, that is, they are not contained inside any object.

Static Function Member Static member functions are not called with respect to an existing object. This function's sole purpose is to access and/or modify static data members of the class.

Static Memory Allocation In this method of memory allocation, the amount of memory to be allocated and the time at which it would get allocated during run time are both decided during compile time itself.

Static Memory Deallocation In this method of memory allocation, the amount of memory to be deallocated and the time at which it would get deallocated during run time are both decided during compile time itself.

Static Polymorphism (see early binding)

Structure A language construct in C language that enables the programmer to put together data that influence each others' values and should therefore be put together. C language does not allow the programmer to define member functions inside structures. This leads to lack of data security.

Subclass (see child class, derived class, inheritance)

SuperClass (see parent class, base class, inheritance)

this pointer A constant pointer that gets passed to each member function as a leading formal argument. It points at the object for which the function is called.

throw A keyword in C++ that allows the library programmer to throw an exception whenever an invalid condition is encountered that cannot be handled in the library code itself. (see also exception handling, try, throw)

try A keyword in C++ that allows a client to place calls to library functions that are likely to throw exceptions. (see also exception handling, try, throw)

typeid Operator An operator that enables us to determine the type of object at which a pointer points.

Virtual Base Class A base class that is derived by using the virtual keyword while declaring the derived class. If a class derives from two classes that in turn inherit from a virtual base class, the final derived class gets only one copy of the features of the virtual base class.

Virtual Function A class member function can be declared as a virtual function by prefixing its declaration with the virtual keyword. Virtual functions enable dynamic polymorphism. If a virtual function is overridden in a derived class and called with respect to a pointer of base class type that points at an object of the same derived class, then the function called would be of the derived class and not of the base class. (see also Dynamic Binding, Dynamic Polymorphism, Late Binding)

Zero-argument Constructor A constructor that does not take any arguments is called a zero-argument constructor. The constructor that is created by default by the compiler is also a zero-argument constructor. Therefore the two terms zero-argument constructor and default constructor are used interchangeably. (see also constructor, default constructor)

Self Tests

Test 1

Time: 1 hour
Max Marks: 50

True/False

[1 × 10 = 10]

1. Variables must be declared at the beginning of the function in a C++ program code.
2. A function can modify the value of the passed parameter if the corresponding formal argument is a reference variable.
3. The presence of an inline function in a code does not impact the size of the resultant executable.
4. Structures in C++ cannot have member functions.
5. A constant member function cannot access the static data members of the class.
6. The return type of a constructor in C++ is `void`.
7. A function of the derived class can access the public member of the base class even if the `private` keyword is used for derivation.
8. A pure virtual function cannot be overloaded.
9. An actual definition is created from a function/class template during run time.
10. Presence of a syntax error in a code will cause an exception to be thrown.

Fill in the Blanks

[1 × 10 = 10]

1. The process of binding together data and code that works upon the data is known as ________________.

2. **cout** is an object of the ________________ class.
3. The ________________ operator is used to access a class member with respect to a pointer.
4. Members declared under the private and ________________ sections of a class cannot be accessed by non-member functions.
5. Virtual functions enable ________________ polymorphism.
6. A virtual function can be specified as a pure virtual function by suffixing its declaration with ________________.
7. Conversion of basic type to class type can be achieved by using ______________.
8. The ________________ flag should be passed to the 'open()' function to ensure that a file does not get created if it does not exist.
9. The three keywords provided by C++ for implementing exception handling are try, catch, and ________________.
10. The syntax for catch all block is ________________.

Multiple Choice Questions

(more than one choice can be correct)

[2 × 5 = 10]

1. Which of the following are features of the Object-Oriented Programming System?
   (a) Inheritance
   (b) Data persistence
   (c) Polymorphism
   (d) Data abstraction
2. Which of the following is a correct function prototype?
   (a) `int abc(int, int);`
   (b) `int abc(int a, int b);`
   (c) `int abc(int a, int b) {}`
   (d) `int abc(int a, b);`
3. The benefits of inheritance is/are
   (a) code reusability
   (b) faster executables
   (c) data hiding
   (d) smaller executables