12

Templates

Key Concepts

- Generic programming
- Multiple parameters in class templates
- Function templates
- Template functions
- Member function templates
- Class templates
- Template classes
- Multiple parameters in class templates
- Overloading of template functions
- Non-type template arguments

12.1 Introduction

Templates is one of the features added to C++ recently. It is a new concept which enable us to define generic classes and functions and thus provides support for *generic programming*. Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.

A template can be used to create a family of classes or functions. For example, a class template for an **array** class would enable us to create arrays of various data types such as **int** array and **float** array. Similarly, we can define a template for a function, say **mul()**, that would help us create various versions of **mul()** for multiplying **int**, **float** and **double** type values.

A template can be considered as a kind of macro. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type. Since a template is defined with a *parameter* that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called *parameterized classes* or *functions*.

12.2 Class Templates

Consider a vector class defined as follows:

```
class vector
{
       int *v;
       int size;
    public:
      vector(int m)            // create a null vector
      {
           v = new int[size = m];
           for(int i=0; i<size; i++)
                v[i] = 0;
      }
      vector(int *a)           // create a vector from an array
      {
           for(int i=0; i<size; i++)
                v[i] = a[i];
      }
      int operator*(vector &y)        // scalar product
      {
      int sum = 0;
      for(int i=0; i<size; i++)
           sum += this -> v[i] * y . v[i];
      return sum;
      }
};
```

The vector class can store an array of **int** numbers and perform the scalar product of two **int** vectors as shown below:

```
int main()
{
     int x[3] = {1,2,3};
     int y[3] = {4,5,6};
     vector v1(3);             // Creates a null vector of 3 integers
     vector v2(3);
     v1 = x;                   // Creates v1 from the array x
     v2 = y;
     int R = v1 * v2;
     cout << "R = " << R;
     return 0;
}
```

Now suppose we want to define a vector that can store an array of **float** values. We can do this by simply replacing the appropriate **int** declarations with **float** in the **vector** class. This means that we have to redefine the entire class all over again.

Assume that we want to define a **vector** class with the data type as a *parameter* and then use this class to create a vector of any data type instead of defining a new class every time. The template mechanism enables us to achieve this goal.

As mentioned earlier, templates allow us to define generic classes. It is a simple process to create a generic class using a template with an anonymous type. The general format of a class template is:

```
template<class T>
class classname
{
      // .....
      // class member specification
      // with anonymous type T
      // wherever appropriate
      // .....
};
```

The template definition of **vector** class shown below illustrates the syntax of a template:

```
template<class T>
class vector
{
        T* v;          // Type T vector
        int size;
  public:
      vector(int m)
      {
           v = new T [size = m];
            for(int i=0; i<size; i++)
              v[i] = 0;
      }
      vector(T* a)
      {
            for(int i=0; i<size, i++)
                 v[i] = a[i];
       }
       T operator*(vector &y)
       {
            T sum = 0;
              for(int i=0; i<size; i++)
                           sum += this -> v[i] * y . v[i];
            return sum;
       }
   };
```

note

The class template definition is very similar to an ordinary class definition except the prefix **template<class T>** and the use of type **T**. This prefix tells the compiler that we are going to declare a template and use **T** as a type name in the declaration. Thus, vector has become a parameterized class with the type **T** as its parameter. **T** may be substituted by any data type including the user-defined types. Now, we can create vectors for holding different data types.

Example:

```
vector <int>   v1(10);        // 10 element int vector
vector <float> v2(25);        // 25 element float vector
```

note

The type **T** may represent a class name as well. Example:

```
vector <complex> v3(5);     // vector of 5 complex numbers
```

A class created from a class template is called a *template class*. The syntax for defining an object of a template class is:

```
classname<type> objectname(arglist);
```

This process of creating a specific class from a class template is called *instantiation*. The compiler will perform the error analysis only when an instantiation takes place. It is, therefore, advisable to create and debug an ordinary class before converting it into a template.

Programs 12.1 and 12.2 illustrate the use of a **vector** class template for performing the scalar product of **int** type vectors as well as **float** type vectors.

Example of Class Template

```
#include <iostream>

using namespace std;

const size = 3;

template <class T>
class vector
{
      T* v;          // type T vector
public:
      vector()
      {
```

(Contd)

```
            v = new T[size];
            for(int i=0;i<size;i++)
                  v[i] = 0;
      }
      vector(T* a)
      {
            for(int i=0;i<size;i++)
                  v[i] = a[i];
      }
      T operator*(vector &y)
      {
            T sum = 0;
            for(int i=0;i<size;i++)
                  sum += this -> v[i] * y.v[i];
            return sum;
      }
};
int main()
{
      int x[3] = {1,2,3};
      int y[3] = {4,5,6};
      vector <int> v1;
      vector <int> v2;
      v1 = x;
      v2 = y;
      int R = v1 * v2;
      cout << "R = " << R << "\n";
      return 0;
  }
```

PROGRAM 12.1

The output of the Program 12.1 would be:

```
R = 32
```

ANOTHER EXAMPLE OF CLASS TEMPLATE

```
#include <iostream>

using namespace std;

const size = 3;
template <class T>
```

(Contd)

```
class vector
{
      T* v;          // type T vector
  public:
      vector()
      {
            v = new T[size];
            for(int i=0;i<size;i++)
                  v[i] = 0;
      }
      vector(T* a)
      {
            for(int i=0;i<size;i++)
                  v[i] = a[i];
      }
      T operator*(vector &y)
      {
            T sum = 0;
            for(int i=0;i<size;i++)
                  sum += this -> v[i] * y.v[i];

            return sum;
      }
};

int main()
{
      float x[3] = {1.1,2.2,3.3};
      float y[3] = {4.4,5.5,6.6};
      vector <float> v1;
      vector <float> v2;
      v1 = x;
      v2 = y;
      float R = v1 * v2;
      cout << "R = " << R << "\n";

      return 0;
 }
```

PROGRAM 12.2

The output of the Program 12.2 would be:

```
R = 38.720001
```

12.3 Class Templates with Multiple Parameters

We can use more than one generic data type in a class template. They are declared as a comma-separated list within the **template** specification as shown below:

```
template<class T1, class T2, ...>
class classname
{
  .....
  .....    (Body of the class)
  .....
};
```

Program 12.3 demonstrates the use of a template class with two generic data types.

TWO GENERIC DATA TYPES IN A CLASS DEFINITION

```
#include <iostream>

using namespace std;

template<class T1, class T2>
class Test
{
        T1 a;
        T2 b;
  public:
        Test(T1 x, T2 y)
        {
                a = x;
                b = y;
        }
        void show()
        {
                cout << a << " and " << b << "\n";
        }
};

int main()
{
        Test <float,int> test1 (1.23,123);
        Test <int,char> test2 (100,'W');

        test1.show();
        test2.show();

        return 0;
};
```

PROGRAM 12.3

The output of Program 12.3 will be would be:

```
1.23 and 123
100 and W
```

12.4 Function Templates

Like class templates, we can also define function templates that could be used to create a family of functions with different argument types. The general format of a function template is:

```
template<class T>
returntype functionname (arguments of type T)
{
        // .....
        // Body of function
        // with type T
        // wherever appropriate
        // .....
}
```

The function template syntax is similar to that of the class template except that we are defining functions instead of classes. We must use the template parameter **T** as and when necessary in the function body and in its argument list.

The following example declares a **swap()** function template that will swap two values of a given type of data.

```
template<class T>
void swap(T&x, T&y)
{
   T temp = x;
   x = y;
   y = temp;
}
```

This essentially declares a set of overloaded functions, one for each type of data. We can invoke the **swap()** function like any ordinary function. For example, we can apply the **swap()** function as follows:

```
void f(int m,int n,float a,float b)
{
     swap(m,n);          // swap two integer values
     swap(a,b);          // swap two float values
     // .....
```

This will generate a **swap()** function from the function template for each set of argument types. Program 12.4 shows how a template function is defined and implemented.

```
#include <iostream>

using namespace std;

template <class T>
void swap(T &x, T &y)
{
     T temp = x;
     x = y;
     y = temp;
}

void fun(int m,int n,float a,float b)
{
     cout << "m and n before swap: " << m << " " << n << "\n";
     swap(m,n);
     cout << "m and n after swap:  " << m << " " << n << "\n";

     cout << "a and b before swap: " << a << " " << b << "\n";
     swap(a,b);
     cout << "a and b after swap:  " << a << " " << b << "\n";
}

int main()
{
     fun(100,200,11.22,33.44);

     return 0;
}
```

PROGRAM 12.4

The output of Program 12.4 would be:

```
m and n before swap:    100 200
m and n after swap:     200 100
a and b before swap:    11.22 33.439999
a and b after swap:     33.439999 11.22
```

Another function often used is **sort()** for sorting arrays of various types such as **int** and **double**.The following example shows a function template for bubble sort:

```
template<class T>
bubble(T v[], int n)
{
      for(int i=0; i<n-1; i++)
      for(int j=n-1; i<j; j--)
            if(v[j] < v[j-1])
            {
                  T temp = v[j];
                  v[j] = v[j-1];
                  v[j-1] = temp;
            }
}
```

Note that the swapping statements

```
T temp = v[j];
v[j] = v[j-1];
v[j-1] = temp;
```

may be replaced by the statement

```
swap(v[j],v[j-1]);
```

where **swap()** has been defined as a function template.

Here is another example where a function returns a value.

```
template<class T>
T max(T x, T y)
{
     return x>y ? x:y;
}
```

A function generated from a function template is called a *template function*. Program 12.5 demonstrates the use of two template functions in nested form for implementing the bubble sort algorithm. Program 12.6 shows another example of application of template functions.

BUBBLE SORT USING TEMPLATE FUNCTIONS

```
#include <iostream>

using namespace std;

template<class T>
void bubble(T a[], int n)
```

(Contd)

```
{
     for(int i=0; i<n-1; i++)
              for(int j=n-1; i<j; j--)
                       if(a[j] < a[j-1])
                       {
                           swap(a[j],a[j-1]);   // calls template function
                       }
}

template<class X>
void swap(X &a, X &b)
{
     X temp = a;
     a = b;
     b = temp;
}

int main()
{
     int x[5] = {10,50,30,40,20};
     float y[5] = {1.1,5.5,3.3,4.4,2.2};

     bubble(x,5);  // calls template function for int values
     bubble(y,5);  // calls template function for float values

     cout << "Sorted x-array: ";
     for(int i=0; i<5; i++)
     cout << x[i] << " ";
     cout << endl;

     cout << "Sorted y-array: ";
     for(int j=0; j<5; j++)
     cout << y[j] << " ";
     cout << endl;

            return 0;
 }
```

PROGRAM 12.5

The output of Program 12.5 would be:

```
Sorted x-array:  10 20 30 40 50
Sorted y-array:  1.1 2.2 3.3 4.4 5.5
```

AN APPLICATION OF TEMPLATE FUNCTION

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

template <class T>
void roots(T a,T b,T c)
{
        T d = b*b - 4*a*c;
        if(d == 0)                      // Roots are equal
        {
                cout << "R1 = R2 = " << -b/(2*a) << endl;
        }
        else if(d > 0)                  // Two real roots
        {
                cout << "Roots are real \n";
                float R = sqrt(d);
                float R1 = (-b+R)/(2*a);
                float R2 = (-b-R)/(2*a);
                cout << "R1 = " << R1 << " and ";
                cout << "R2 = " << R2 << endl;
        }
        else                            // Roots are complex
        {
                cout << "Roots are complex \n";
                float R1 = -b/(2*a);
                float R2 = sqrt(-d)/(2*a);
                cout << "Real part = " << R1 << endl;
                cout << "Imaginary part = " << R2;
                cout << endl;
        }
}

int main()
{
        cout << "Integer coefficients \n";
        roots(1,-5,6);
        cout << "\nFloat coefficients \n";
        roots(1.5,3.6,5.0);

        return 0;
}
```

PROGRAM 12.6

The output of Program 12.6 would be:

```
Integer coefficients
Roots are real

R1 = 3 and R2 = 2

Float coefficients
Roots are complex
Real part = -1.2
Imaginary part = 1.375985
```

12.5 Function Templates with Multiple Parameters

Like template classes, we can use more than one generic data type in the template statement, using a comma-separated list as shown below:

```
template<class T1, class T2, ...>
returntype functionname(arguments of types T1, T2,...)
{
        .....
        .....   (Body of function)
        .....
}
```

Program 12.7 illustrates the concept of using two generic types in template functions.

FUNCTION WITH TWO GENERIC TYPES

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

using namespace std;

template<class T1, class T2>
void display(T1 x, T2 y)
{
     cout << x << " " << y << "\n";
}

int main()
{
     display(1999, "EBG");
     display(12.34, 1234);
     return 0;
}
```

PROGRAM 12.7

The output of Program 12.7 would be:

```
1999 EBG
12.34 1234
```

12.6 Overloading of Template Functions

A template function may be overloaded either by template functions or ordinary functions of its name. In such cases, the overloading resolution is accomplished as follows:

1. Call an ordinary function that has an exact match.
2. Call a template function that could be created with an exact match.
3. Try normal overloading resolution to ordinary functions and call the one that matches.

An error is generated if no match is found. Note that no automatic conversions are applied to arguments on the template functions. Program 12.8 shows how a template function is overloaded with an explicit function.

TEMPLATE FUNCTION WITH EXPLICIT FUNCTION

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

using namespace std;

template <class T>
void display(T x)
{
     cout << "Template display: " << x << "\n";
}
void display(int x)        // overloads the generic display()
{
     cout << "Explicit display: " << x << "\n";
}

int main()
{
     display(100);
     display(12.34);
     display('C');

     return 0;
}
```

PROGRAM 12.8

The output of Program 12.8 would be:

```
Explicit display: 100
Template display: 12.34
Template display: C
```

note

The call **display(100)** invokes the ordinary version of **display()** and not the template version.

12.7 Member Function Templates

When we created a class template for vector, all the member functions were defined as inline which was not necessary. We could have defined them outside the class as well. But remember that the member functions of the template classes themselves are parameterized by the type argument (to their template classes) and therefore these functions must be defined by the function templates. It takes the following general form:

```
Template<class T>
returntype classname <T> :: functionname(arglist)
{
        // .....
        // Function body
        // .....
}
```

The **vector** class template and its member functions are redefined as follows:

```
// Class template ..........

template<class T>
class vector
{
          T* v;
          int size;
  public:
      vector(int m);
       vector(T* a);
        T operator*(vector & y);
};

// Member function templates .........
template<class T>
```

```
vector<T> :: vector(int m)
{
     v = new T[size = m];
     for(int i=0; i<size; i++)
         v[i] = 0;
}

template< class T>
vector<T> :: vector(T* a)
{
     for(int i=0; i<size; i++)
          v[i] = a[i];
}

template< class T>
T vector<T> :: operator*(vector & y)
{
     T sum = 0;
     for(int i = 0; i < size; i++)
     sum += this -> v[i] * y.v[i];
     return sum;
}
```

12.8 Non-Type Template Arguments

We have seen that a template can have multiple arguments. It is also possible to use non-type arguments. That is, in addition to the type argument **T**, we can also use other arguments such as strings, function names, constant expressions and built-in types. Consider the following example:

```
template<class T, int size>
class array
{
     T a[size];          // automatic array initialization
     // .....
     // .....
};
```

This template supplies the size of the **array** as an argument. This implies that the size of the **array** is known to the compiler at the compile time itself. The arguments must be specified whenever a template class is created. Example:

```
array<int,10>  a1;      // Array of 10 integers
array<float,5> a2;      // Array of 5 floats
array<char,20> a3;      // String of size 20
```

The size is given as an argument to the template class.

SUMMARY

- C++ supports a mechanism known as template to implement the concept of generic programming.
- Templates allows us to generate a family of classes or a family of functions to handle different data types.
- Template classes and functions eliminate code duplication for different types and thus make the program development easier and more manageable.
- We can use multiple parameters in both the class templates and function templates.
- A specific class created from a class template is called a template class and the process of creating a template class is known as instantiation. Similarly, a specific function created from a function template is called a template function.
- Like other functions, template functions can be overloaded.
- Member functions of a class template must be defined as function templates using the parameters of the class template.
- We may also use non-type parameters such basic or derived data types as arguments templates.

Key Terms

- bubble sort
- class template
- **display()**
- **explicit** function
- function template
- generic programming
- instantiation
- member function template
- multiple parameters
- overloading
- **parameter**
- parameterized classes
- parameterized functions
- swapping
- **swap()**
- **template**
- template class
- template definition
- template function
- template parameter
- template specification
- templates

Review Questions

12.1 *What is generic programming? How is it implemented in C++?*

12.2 *A template can be considered as a kind of macro. Then, what is the difference between them?*

12.3 *Distinguish between overloaded functions and function templates.*

12.4 *Distinguish between the terms class template and template class.*

12.5 *A class (or function) template is known as a parameterized class (or function). Comment.*

12.6 *State which of the following definitions are illegal.*

(a)
```
template<class T>
class city
{ ..... };
```

(b)
```
template<class P, R, class S>
class city
{ ..... }
```

(c)
```
template<class T, typename S>
class city
{ ..... };
```

(d)
```
template<class T, typename S>
class city
{ ..... };
```

(e)
```
class<class T, int size=10>
class list
{ ..... };
```

(f)
```
class<class T = int, int size>
class list
{ ..... };
```

12.7 *Identify which of the following function template definitions are illegal.*

(a)
```
template<class A, B>
void fun(A, B)
{ ..... };
```

(b)
```
template<class A, class A>
void fun(A, A)
{ ..... };
```

(c)
```
template<class A>
void fun(A, A)
{ ..... };
```

(d)
```
template<class T, typename R>
T fun(T, R)
{ ..... };
```
(e)
```
template<class A>
A fun(int *A)
{ ..... };
```

Debugging Exercises

12.1 Identify the error in the following program.

```
#include <iostream.h>
class Test
{
        int intNumber;
        float floatNumber;
public:
        Test()
        {
                intNumber = 0;
                floatNumber = 0.0;
        }

        int getNumber()
        {
                return intNumber;
        }

        float getNumber()
        {
                return floatNumber;
        }
};

void main()
{
        Test objTest1;
        objTest1.getNumber();
}
```

12.2 Identify the error in the following program.

```
#include <iostream.h>

template <class T1, class T2>
```

13

Exception Handling

Key Concepts

- Errors and exceptions
- Throwing mechanism
- Multiple catching
- Rethrowing exceptions
- Exception handling mechanism
- Catching mechanism
- Catching all exceptions
- Restricting exceptions thrown

13.1 Introduction

We know that it is very rare that a program works correctly first time. It might have bugs. The two most common types of bugs are *logic errors* and *syntactic errors*. The logic errors occur due to poor understanding of the problem and solution procedure. The syntactic errors arise due to poor understanding of the language itself. We can detect these errors by using exhaustive debugging and testing procedures.

We often come across some peculiar problems other than logic or syntax errors. They are known as *exceptions*. Exceptions are run time anomalies or unusual conditions that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory or disk space. When a program encounters an exceptional condition, it is important that it is identified and dealt with effectively. ANSI C++ provides built-in language features to detect and handle exceptions which are basically run time errors.

Exception handling was not part of the original C++. It is a new feature added to ANSI C++. Today, almost all compilers support this feature. C++ exception handling provides a

type-safe, integrated approach, for coping with the unusual predictable problems that arise while executing a program.

13.2 Basics of Exception Handling

Exceptions are of two kinds, namely, *synchronous exceptions* and *asynchronous exceptions*. Errors such as "out-of-range index" and "over-flow" belong to the synchronous type exceptions. The errors that are caused by events beyond the control of the program (such as keyboard interrupts) are called asynchronous exceptions. The proposed exception handling mechanism in C++ is designed to handle only synchronous exceptions.

The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstance" so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks:

1. Find the problem (*Hit the exception*).
2. Inform that an error has occurred (*Throw the exception*).
3. Receive the error information (*Catch the exception*).
4. Take corrective actions (*Handle the exception*).

The error handling code basically consists of two segments, one to detect errors and to throw exceptions, and the other to catch the exceptions and to take appropriate actions.

13.3 Exception Handling Mechanism

C++ exception handling mechanism is basically built upon three keywords, namely, **try**, **throw**, and **catch**. The keyword **try** is used to preface a block of statements (surrounded by braces) which may generate exceptions. This block of statements is known as *try block*. When an exception is detected, it is thrown using a **throw** statement in the try block. A *catch block* defined by the keyword **catch** 'catches' the exception 'thrown' by the throw statement in the try block, and handles it appropriately. The relationship is shown in Fig. 13.1.

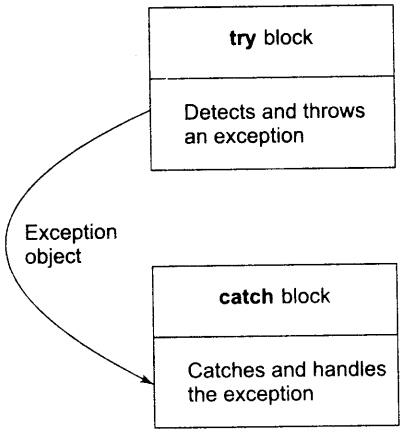

Fig. 13.1 ⇔ *The block throwing exception*

The **catch** block that catches an exception must immediately follow the **try** block that throws the exception. The general form of these two blocks are as follows:

```
.....
.....
try
{
        .....
        throw exception;            // Block of statements which
        .....                       // detects and throws an exception
        .....
}
catch(type arg)                     // Catches exception
{
        .....
        .....                       // Block of statements that
        .....                       // handles the exception
        .....
}
.....
.....
```

When the **try** block throws an exception, the program control leaves the **try** block and enters the **catch** statement of the catch block. Note that exceptions are objects used to transmit information about a problem. If the type of object thrown matches the *arg* type in the **catch** statement, then catch block is executed for handling the exception. If they do not match, the program is aborted with the help of the **abort()** function which is invoked by default. When no exception is detected and thrown, the control goes to the statement immediately after the catch block. That is, the catch block is skipped. This simple try-catch mechanism is illustrated in Program 13.1.

TRY BLOCK THROWING AN EXCEPTION

```
#include <iostream>

using namespace std;
int main()
{
    int a,b;
    cout << "Enter Values of a and b \n";
    cin >> a;
    cin >> b;
    int x = a-b;
    try
    {
        if(x != 0)
        {
```

(Contd)

```
            cout << "Result(a/x) = " << a/x << "\n";
        }
        else                    // There is an exception
        {
                throw(x);    // Throws int object
        }
    }
    catch(int i)                // Catches the exception
    {
            cout << "Exception caught: x = " << x << "\n";
    }

    cout << "END";

    return 0;
}
```

PROGRAM 13.1

The output of Program 13.1:

First Run

```
Enter Values of a and b
20 15
Result(a/x) = 4
END
```

Second Run

```
Enter Values of a and b
10 10
Exception caught: x = 0
END
```

Program detects and catches a division-by-zero problem. The output of first run shows a successful execution. When no exception is thrown, the **catch** block is skipped and execution resumes with the first line after the **catch**. In the second run, the denominator x becomes zero and therefore a division-by-zero situation occurs. This exception is thrown using the object **x**. Since the exception object is an **int** type, the **catch** statement containing **int** type argument catches the exception and displays necessary message.

Most often, exceptions are thrown by functions that are invoked from within the **try** blocks. The point at which the **throw** is executed is called the *throw point*. Once an exception is thrown to the catch block, control cannot return to the throw point. This kind of relationship is shown in Fig. 13.2.

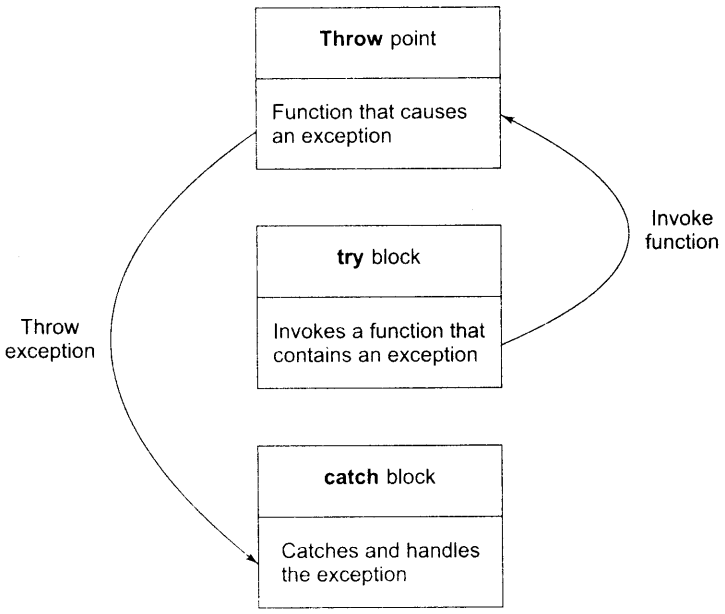

Fig. 13.2 ⇔ *Function invoked by try block throwing exception*

The general format of code for this kind of relationship is shown below:

```
type function(arg list)      // Function with exception
{
        ......
        ......
        throw(object);       // Throws exception
        ......
        ......
}
......
......
try
{
        ......
        ...... Invoke function here
        ......
}
catch(type arg)              // Catches exception
{
        ......
        ...... Handles exception here
        ......
}
......
```

note

The **try** block is immediately followed by the **catch** block, irrespective of the location of the throw point.

Program 13.2 demonstrates how a **try** block invokes a function that generates an exception.

INVOKING FUNCTION THAT GENERATES EXCEPTION

```
// Throw point outside the try block

#include <iostream>

using namespace std;

void divide(int x, int y, int z)
{
        cout << "\nWe are inside the function \n";
        if((x-y) != 0)          // It is OK
        {
              int R = z/(x-y);
              cout << "Result = " << R << "\n";
        }
        else                    // There is a problem
        {
              throw(x-y);       // Throw point
        }
}

int main()
{
        try
        {
              cout << "We are inside the try block \n";
              divide(10,20,30);   // Invoke divide()
              divide(10,10,20);   // Invoke divide()
        }
        catch(int i)      // Catches the exception
        {
              cout << "Caught the exception \n";
        }
        return 0;
}
```

PROGRAM 13.2

Program 13.3 shows a simple example where multiple catch statements are used to handle various types of exceptions.

MULTIPLE CATCH STATEMENTS

```
#include <iostream>

using namespace std;

void test(int x)
{
        try
        {
                if(x == 1) throw x;                    // int
                else
                        if(x == 0) throw 'x';          // char
                else
                        if(x == -1) throw 1.0;         // double
                cout << "End of try-block \n";
        }
        catch(char c) // Catch 1
        {
                cout << "Caught a character \n";
        }
        catch(int m)  // Catch 2
        {
                cout << "Caught an integer \n";
        }
        catch(double d)       // Catch 3
        {
                cout << "Caught a double \n";
        }
        cout << "End of try-catch system \n\n";
}

int main()
{
        cout << "Testing Multiple Catches \n";
        cout << "x ==  1 \n";
        test(1);
        cout << "x ==  0 \n";
        test(0);
        cout << "x ==  1 \n";
        test(-1);
        cout << "x ==  2 \n";
        test(2);

        return 0;
   }
```

PROGRAM 13.3

The output of the Program 13.3:

```
Testing Multiple Catches
x == 1
Caught an integer
End of try-catch system

x == 0
Caught a character
End of try-catch system

x == -1
Caught a double
End of try-catch system0

x == 2
End of try-block
End of try-catch system
```

The program when executed first, invokes the function **test()** with x = 1 and therefore throws x an **int** exception. This matches the type of the parameter **m** in catch2 and therefore catch2 handler is executed. Immediately after the execution, the function **test()** is again invoked with x = 0. This time, the function throws 'x', a character type exception and therefore the first handler is executed. Finally, the handler catch3 is executed when a double type exception is thrown. Note that every time only the handler which catches the exception is executed and all other handlers are bypassed.

When the **try** block does not throw any exceptions and it completes normal execution, control passes to the first statement after the last **catch** handler associated with that try block.

note

try block does not throw any exception, when the **test()** is invoked with x = 2.

Catch All Exceptions

In some situations, we may not be able to anticipate all possible types of exceptions and therefore may not be able to design independent **catch** handlers to catch them. In such circumstances, we can force a **catch** statement to catch all exceptions instead of a certain type alone. This could be achieved by defining the **catch** statement using ellipses as follows:

```
catch(...)
{
        // Statements for processing
        // all exceptions
}
```

```
        try
        {
                divide(10.5,2.0);
                divide(20.0,0.0);
        }
        catch(double)
        {
                cout << "Caught double inside main \n";
        }
        cout << "End of main \n";

        return 0;
}
```

PROGRAM 13.5

The output of the Program 13.5:

```
Inside main
Inside function
Division = 5.25
End of function

Inside function
Caught double inside function
Caught double inside main
End of main
```

When an exception is rethrown, it will not be caught by the same **catch** statement or any other **catch** in that group. Rather, it will be caught by an appropriate **catch** in the outer **try/catch** sequence only.

A **catch** handler itself may detect and throw an exception. Here again, the exception thrown will not be caught by any **catch** statements in that group. It will be passed on to the next outer **try/catch** sequence for processing.

13.7 Specifying Exceptions

It is possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a **throw** *list* clause to the function definition. The general form of using an *exception specification* is:

```
type function(arg-list) throw (type-list)
{
        ......
        ...... Function body
        ......
}
```

The *type-list* specifies the type of exceptions that may be thrown. Throwing any other type of exception will cause abnormal program termination. If we wish to prevent a function from throwing any exception, we may do so by making the *type-list* empty. That is, we must use

```
throw(); // Empty list
```

in the function header line.

note

A function can only be restricted in what types of exceptions it throws back to the *try* block that called it. The restriction applies only when throwing an exception out of the function (and not within a function).

Program 13.6 demonstrates how we can restrict a function to throw only certain types and not all.

TESTING THROW RESTRICTIONS

```
#include <iostream>

using namespace std;

void test(int x) throw(int,double)
{
        if(x == 0) throw 'x';                  // char
        else
                if(x == 1) throw x;            // int
        else
                if(x == -1) throw 1.0;         // double
        cout << "End of function block \n";
}

int main()
{
        try
        {
                cout << "Testing Throw Restrictions \n";
                cout << "x ==  0 \n";
                test(0);
                cout << "x ==  1 \n";
                test(1);
                cout << "x == -1 \n";
                test(-1);
                cout << "x ==  2 \n";
```

(Contd)

13.10 *State what will happen in the following situations:*

(a) *An exception is thrown outside a* ***try*** *block*
(b) *No* ***catch*** *handler matches the type of exception thrown*
(c) *Several handlers match the type of exception thrown*
(d) *A* ***catch*** *handler throws an exception*
(e) *A function throws an exception of type not specified in the specification list*
(f) ***catch(...)*** *is the first of cluster of* ***catch*** *handlers*
(g) *Placing* ***throw()*** *in a function header line*
(h) *An exception rethrown within a* ***catch*** *block*

13.11 *Explain under what circumstances the following statements would be used:*

(a) `throw;`
(b) `void fun1(float x) throw()`
(c) `catch(...)`

Debugging Exercises

13.1 Identify the error in the following program.

```
#include <iostream.h>
class Person
{
      int age;
public:
      Person()
      {
      }

      Person(int i):age(i)
      {
      }

      void getOccupation()
      {
            try
            {
                        switch(age)
                        {
                        case 10:
                              throw ("Child");
                        case 20:
                              throw "Student";
```

```
                                    break;
                            case 30:
                                    throw "Employee";
                                    break;
                            }
                    }

            }
            void operator ++()
            {
                    age+=10;
            }
};
void main()
{
            Person objPerson(10);
            objPerson.getOccupation();
            ++objPerson;
            objPerson.getOccupation();
            ++objPerson;
            objPerson.getOccupation();
}
```

13.2 Identify the error in the following program.

```
#include <iostream.h>

void callFunction(int i)
{
        if(i)
                    throw 1;
        else
                    throw 0;
}

void callFunction(char *n)
{
          try
          {
                    if(n)
                            throw "StringOK";
```

```
                else
                        throw "StringError";
        }
        catch(char* name)
        {
                cout << name << " ";
        }
}

void main()
{
        try
        {
             callFunction("testString");
             callFunction(1);
             callFunction(0);
        }
        catch(int i)
        {
             cout << i << " " ;
        }
        catch(char *name)
        {
             cout << name << " ";
        }
}
```

13.3 Identify the error in the following program.

```
#include <iostream.h>

class Mammal
{
public:
      Mammal()
      {
      }

      class Human
      {
```

```
        };

        class Student  : virtual public Human
        {
        };

        class Employee : virtual public Human
        {
        };

        void getObject()
        {
             throw Employee();
        }
};
void main()
{
        Mammal m;
        try
        {
             m.getObject();
        }
        catch(Mammal::Human&)
        {
             cout << "Human ";
        }
        catch(Mammal::Student&)
        {
             cout << "Student ";
        }
        catch(Mammal::Employee&)
        {
             cout << "Employee ";
        }
        catch(...)
        {
             cout << "All";
        }
}
```

13.4 Identify errors, if any, in the following statements.

(a)
```
catch(int a, float b)
{...}
```
(b)
```
try
{throw 100;};
```
(c)
```
try
{fun1()}
```
(d)
```
throw a, b;
```
(e)
```
void divide(int a, int b) throw(x, y)
{......}
```
(f)
```
catch(int x, ..., float y)
{......}
```
(g)
```
try
{throw x/y;}
```
(h)
```
try
{if(!x) throw x;}
catch(x)
{cout << "x is zero \n";}
```

Programming Exercises

13.1 *Write a program containing a possible exception. Use a try block to throw it and a catch block to handle it properly.*

13.2 *Write a program that illustrates the application of multiple catch statements.*

13.3 *Write a program which uses* ***catch(...)*** *handler.*

13.4 *Write a program that demonstrates how certain exception types are not allowed to be thrown.*

13.5 *Write a program to demonstrate the concept of rethrowing an exception.*

13.6 *Write a program with the following:*

(a) *A function to read two double type numbers from keyboard*

(b) *A function to calculate the division of these two numbers*

(c) *A try block to throw an exception when a wrong type of data is keyed in*

(d) *A try block to detect and throw an exception if the condition "divide-by-zero" occurs*

(e) *Appropriate catch block to handle the exceptions thrown*

13.7 *Write a main program that calls a deeply nested function containing an exception. Incorporate necessary exception handling mechanism.*