TEMPLATES

TEMPLATES

Templates are a relatively new addition to C++. They allow you to write generic classes and functions that work for several different data types. The result is that you can write generic code once and then use it over again for many different uses. C++ uses the templates to enable generic techniques. Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations.

In C++, class and function templates are particularly effective mechanisms for generic programming because they make the generalization possible without sacrificing efficiency.

FUNCTION TEMPLATES

```
int sum( int a, int b)
{
 return (a+b);
}
```

The above code is an example of a normal function. It takes two integer values, sums them and returns the sum. However, If we try to call the sum function as sum(2.3,4.5), the compiler will throw an error as the sum function is designed to work on integer values only. We cannot pass other data types to the function. To make the above function flexible, C++ has introduced the concept of templates.

In C++, function overloading allows defining multiple functions with the same names but with different arguments. In a situation, where we need to define functions with the same body but of different data types we can use function overloading. But while overloading a function, different data types will need different functions, and hence the function needs to be rewritten several times for each data type. This is time consuming and space consuming.

FUNCTION TEMPLATES

So C++ introduced the new concept of function template. Function templates are special functions that can operate with generic types. It is the function which is written just for once, and have it worked for many different data type. The key innovation in function templates is to represent the data type used by the function not as a specific type such as int, float, etc., but as a name that can stand for any type.

```
template <class T>
return_type function_name(T type argument){
// body
 }
```

The template keyword informs the compiler that we're about to define a function template. The keyword class, within angle brackets, acts as the type. We can define our own data type using classes, so there's no distinction between types and classes. The variable following keyword class ('T' in above syntax) is called the template argument.

FUNCTION TEMPLATES

```
#include <iostream>
using namespace std;
template <class T1>
T1 sum(T1 a, T1 b)
{
    return (a+b);
}
```

```
int main()
{
    cout << sum(10, 20) << endl; //calling the function the function template sum for the integer values
    cout << sum(11.5, 13.1) << endl; //calling the function the function template sum for the float values

    return 0;
}
```

In the above example, sum() is a template function. It takes two arguments of T1 type and returns a T1 type data equal to the sum of two arguments. Defining a function as a template doesn’t generate code for different types, it is just a message to the compiler that the function can be called with different data types. Thus, in main(), when sum(10,20) is called, the template generates a function by substituting template argument (T1) with int. This is called instantiating the function template, and each instantiating version of the function is called template function. When sum(11.5,13.1) is invoked, instantiating of the template by float takes place and another template function of float type is created. Here the function template sum can take arguments of any type(int, float,char etc....)

FUNCTION TEMPLATE WITH MULTIPLE ARGUMENTS

We can use more than one generic type in a function template. They are declared as a comma - separated list within the template specification.

```
template <class T1, class T2, ...>
returntype functionname(arguments of type T1, T2, T3...)
{
  // body
}
```

OVERLOADING TEMPLATES

Overloading with function templates

```
#include<iostream>
using namespace std;
//template
template<class T>
T find_max( T a, T b)
{
   if(a>b)
      return a;
   else
      return b;
}
```

```
//normal function having same name as template
float find_max(float x, float y)
{
      if(x>y)
     return x;
     else
     return y;
}
```

```
int main()
{
   int p = 15, q = 12,r;
   float c = 18.9, d = 25.009,e;
 r = find_max(p,q); //template find_max is called.
e = find_max(c,d); //normal function find_max is called.
   cout<<"The greatest integer value is "<<r<<endl;
   cout<<"The greatest float value is "<<e;
   return 0;
}
```

Output:

The greatest integer value is 15

The greatest float value is 25.009

In the above example, the find_max() function is overloaded. Two functions of same name and same arguments are declared, one as a function template and other of type float. The function template generates functions for every data type but nontemplate functions take precedence over template function. Thus, find_max(c,d) invokes normal function as c and d are of type float and find_max(x,y) invokes template function.

OVERLOADING TEMPLATES

Overloading with other template

```
#include<iostream>
using namespace std;
template<class T>
void display( T p )
{
   cout<< p <<endl;
}
/ overloading template with other template having two arguments
template<class T>
void display( T p, int q)
{
 for(int i = 0;i < q; i++)
 cout<< p <<endl;
}
```

```
int main()
{
   display(5); // the first template "display" is called for integer value.
 display("hey", 3); //the second template "display" is called for string.
display(10.8,2); //the second template "display" is called for float value.
}
```

Output:

```
5
hey
hey
hey
10.8
10.8
```

The above example has two template functions that are overloaded. The display() function chosen depends on number of arguments as the two functions are distinguished by the number of arguments. Thus display(5) uses the first template function whereas display("hey", 3) and display(10.8,2) uses the second template function.

CLASS TEMPLATES

Class template is a template used to generate template classes. Class templates are used to construct classes having the same functionality for different data types. A class template provides specification for generating classes based on parameters.

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

Template class is an instance of a class template as a template function is an instance of a function template.

Syntax(for defining the object of template class):

```
classname<type> objectname;
```

CLASS TEMPLATES

Function Definition of Class Template

If the functions are defined outside the template class body, they should always be defined with the full template definition. They must be written like normal functions with scope resolution operator.

```
template <class T>
returntype classname <T>::functionname(arglist)
{
 // body of function
 }
```

CLASS TEMPLATES

Function Definition of Class Template

```
#include<iostream>
using namespace std;
template<class T1>
class numbers
{
   T1 a, b;
   public:
 numbers( T1 x, T1 y)
   {
      a = x;
      b = y;
   }
 T1 getmax();
};
```

```
template<class T>
T numbers<T>::getmax() //getmax() must be defined outside the class using full template definition
{
   if(a>b)
   {
      return a;
   }
   e  l  s  e
   {
      return b;
   }

int main()
{
 numbers<int> obj1(5,9); //obj1 is of type int
 numbers<float> obj2(3.4,1.5); //obj2 is of type float
int x = obj1.getmax();
float y = obj2.getmax();
cout<<"The greatest integer value is "<<x<<endl;
cout<<"The greatest float value is "<<y<<endl;
return 0;
}
```

Output:

The greatest integer value is 9

The greatest float value is 3.4

CLASS TEMPLATES

Class Template with multiple arguments

Like in function template, we can use more than one generic type in class template. Each type are declared in Template specification, and they are separated by comma operator.

Syntax (for defining class template):

template <class T1, class T2, ... >
class classname
{
//body of class
};

Syntax (for defining object of the class):

template <type1, type2, ... > objectname;

Syntax (for defining member function outside class):

template <class T1, class T2, ... >
returntype classname<T1, T2, ...>::functionname (argumentlist)
{
// body of function
}

CLASS TEMPLATES

Function Definition of Class Template

```
#include <iostream> using
namespace std;
template<class T1, class T2>
class Record
{
T1 a;
T2 b;
public:
Record(T1 x, T2 y)
 {
 a = x;
b = y;
 }
 void show();
};
```

```
template<class T1, class T2>
void Record<T1,T2>::show()
{
 cout << a << " and " << b << endl;
}
int main()
{
Record<int, char>Obj1(1,'A');
Record<float, char> Obj2(12.3,'B');
Obj1.show();
Obj2.show();
return 0;
}
```

Output:

1 and A

12.3 and B

The class ‘Record’ can take two generic types to store the records of the two types. Obj1 stores an int and a char and Obj2 stores a float and a char.

NON-TEMPLATE TYPE ARGUMENTS

A nontype template argument provided within a template argument list is an expression whose value can be determined at compile time. Such arguments must be constant expressions. addresses of functions or objects with external linkage, or addresses of static class members. Nontemplate arguments are normally used to initialise a class or to specify the sizes of class members.

Syntax:

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

NON-TEMPLATE TYPE ARGUMENTS

```
#include <iostream>
using namespace std;
template <class T, int s>
class Array
{
      T a[s];
      public:
           void input()
           {
                 cout << "enter numbers " << endl;
                for(int i=0; i<s; i++)
                 cin >> a[i];
            }
           void display();

};
```

```
template <class T, int s>
void Array<T, s>::display()
{
      for(int i=0; i<s; i++)
      cout << a[i] << endl;
}
int main()
{
      Array<int,4>a1;
      a1.input();
      a1.display();
      Array<float,3>a2;
      a2.input();
      a2.display();
      return 0;
}
```

Output:

```
enter  numbers
2 3 4 5
2
3
4
5
enter  numbers
2.5 3.2 2
2.5
3.2
2
```

NON-TEMPLATE TYPE ARGUMENTS

Default Arguments with Class Template

Template parameters may have default arguments. The set of default template arguments accumulates over all declarations of a given template. The following example demonstrates this:

```
template<class T = int, int size = 10>
class Array
{
   //body
};
int main()
{
 //....
 Array<float, 5> ftarray; //float array with size 5
//....
Array<double> darray; //double array with the default size of 10
//....
Array<> intarr; //default integer array with the default size of 10
//....
}
```

NON-TEMPLATE TYPE ARGUMENTS

Default Arguments with Class Template

```
// WAP to show the implementation of default argument with class template

 #include <iostream>
 using namespace std;
 template <class T=float, int s=2>
class Array
{

     T          a[s];
    public:
    void  input();
    void sum();

};
template <class T, int s>
void Array<T,s>::input()
{

     cout << "enter numbers " << endl;
    for(int i=0; i<s; i++)
    cin >> a[i];

}
```

```
template <class T, int s>
void Array<T, s>::sum()
{
     T sum = 0;
    for(int i=0; i<s; i++)
     sum += a[i];
     cout << "The sum is : " << sum << endl;
 }
 int main()
 {
    cout << "For integer array of size 5" << endl;
    Array<int,5> a1;
    a1.input();
    a1.sum();
    cout << "For default values" << endl;
    Array<>a2; //by default a2 object contains a float array with the size of 2
    a2.input();
    a2.sum();

     return 0;
}
```

Output:

```
enter numbers
1 2 3 4 5
The sum is : 15
For  default  values
enter numbers
1.1
2.2
The sum is : 3.3
```

DERIVED CLASS TEMPLATE

1. We can create a derived class which is a non-template class from a base class which is a template class.
2. We can create a derived class which is a template class from a base class which is not a template class.
3. We can create derived class which is a template class from a base class which is also a template class with the same template parameters as in the base class
4. We can create a derived class which is a template class from a base class which is also a template class with additional template parameters in the derived class than that of the class

DERIVED CLASS TEMPLATE

1. If we don't add extra template parameter and supply the template argument of base class with data type, we create a non-template derived class as:

```
#include<iostream>
using namespace std;
template <class T>
class base
{
 T data;
 public:
 base(){}  base(T a){data = a;}
void display()
{
cout<<"data: "<<data<<endl;

 }

};
```

```
class derived1: public base<int>
{
 public:
 derived1(){}
derived1(int a): base<int>(a){}
};
int main()
 {
 derived1 obj1(5);
 obj1.display();
}
```

DERIVED CLASS TEMPLATE

2. If we add extra template parameter and supply the template argument of base class with data type, we create a derived class template as:

```
#include<iostream>
using namespace std;
template <class T>
class base
{
 T data;
 public:
 base(){}
base(T a)
{data = a;}
void display()
{
cout<<”data: ”<<data<<endl;
 }
};
```

```
template <class T>
class derived2: public base<int>
{
 public:
 derived2(){}
derived2(int a, T b):
base<int>(a),data(b){}
void display()
{
cout<<”in base”;
base<int>::display();
cout<<””in derived, data: ”<<data<<endl;  }
};
int main()
{
 derived2<float> obj2(10, 12.34);
 obj2.display();
}
```

DERIVED CLASS TEMPLATE

3. If the base class template parameter is still useful in derived class, the derived class is created as class template i.e., base and derived template classes have the same template parameter.

```
#include<iostream>
using namespace std;
template <class T>
class base
{
 T data;
 public:
 base(){}
base(T a){data = a;}
void display()
{
cout<<"data:"<<data<<endl;
 }

};
```

```
template <class T>
class derived3: public base<T>
{
 public:
derived3(){}
derived3(T a): base<T>(a){}
};
int main()
{
 derived3<int> obj3(5);
 obj3.display();
}
```

DERIVED CLASS TEMPLATE

4. We can also add an extra template parameter in the derived class along with the base class template parameter

```
#include<iostream>
using namespace std;
template <class T>
class base
{
 T data;
 public:
 base(){}
base(T a){data = a;}
void display()
{
cout<<”data: ”<<data<<endl;
 }

};
```

```
template <class T1, class T2>
class derived4: public base<T1>
{
 T2 data;
 public:
derived4(){}
derived4(T1  a,  T2  b):  base<T1>(a),data(b){}
void display()
{
cout<<”in base”;
base<T1>::display();
cout<<””in derived, data: ”<<data<<endl;
}
};
int main()
{
 derived4<int. float> obj4(10, 12.34);
 obj4.display();
}
```

DERIVED CLASS TEMPLATE

5. The derived class template can be created from the base class which is not a class template. In this case, a template parameter is added in the derived class during inheritance.

```
#include<iostream>
using namespace std;
class base
{
 int data;
 public:
 base(){}
base(int a){data = a;}
void display()
{
cout<<"data: "<<data<<endl;
 }

};
```

```
template <class T> class
derived5: public base
{
 public:
 derived5(){}
derived5(int a, T b): base(a),data(b){}
void display()
{
cout<<"in base";
base::display();
cout<<""in derived, data: "<<data<<endl;
}
};
int main()
{
 derived5<float> obj5(25, 10.5);
 obj5.display();
}
```

1. What is a function template?

a. A function that can work with multiple data types
b. A function that is used to create a new classes
c. A function that can take any number of arguments
d. A function that is used to create a new functions

2. Which keyword is used to define a function template in C++?

a. Function
b. Template
c. Typename
d. None

3. Which symbol is used to specify the template parameter in a function template?

a. &
b. *
c. <>
d. None

4. What is the syntax for defining a function template in C++?

a. Template <class T> void functionName(T arg)
b. Void functionName(T arg) template <typename T>
c. Template <typename T> void functionName(T arg)
d. None

5. What is the syntax for overloading function template in C++?

a. Template <typename T> void functionName(T arg) template <typename T, typename U> void functionName(T arg, U u)
b. Template <typename T> void functionName(T arg) void functionName(T arg, Int n)
c. Template <typename T, typename U> void functionName(T arg, U u) template <typename T> void functionName(T arg)
d. None

6. What is the purpose of function template specialization in C++?

a. It allows for the template to be more flexible
b. It allows for the template to handle a specific data type differently
c. It allows for the template to be more efficient
d. None

7. How do you declare an instance of a clas template in C++?

a. ClassName objectName <T>
b. T className objectName
c. className <T> objectName
d. None

8. What is the syntax for class template specialization in C++?

a. Template <T> class className <> {}
b. Class <> template className <T> {}
c. Template <> class className <T> {}
d. None

9. What is the syntax for defining a member function of a class template in C++?

a. Template <typename T> void ClassName<T>::void functionName(){}
b. ClassName<T>::template void functionName(){}
c. Template <typename T> void ClassName<T>::functionName(){}
d. None

10. What is the syntax for calling a function template in C++?

a. FunctionName<N>(parameter);
b. FunctionName<T>(parameter);
c. FunctionName<T…>(parameter);
d. None

11. What is the syntax for calling a member function of a class template in C++?

a. ObjectName.functionName<T>();
b. ObjectName.functionName<N>();
c. ObjectName.functionName<T…>();
d. None

12. What is a function template in C++?

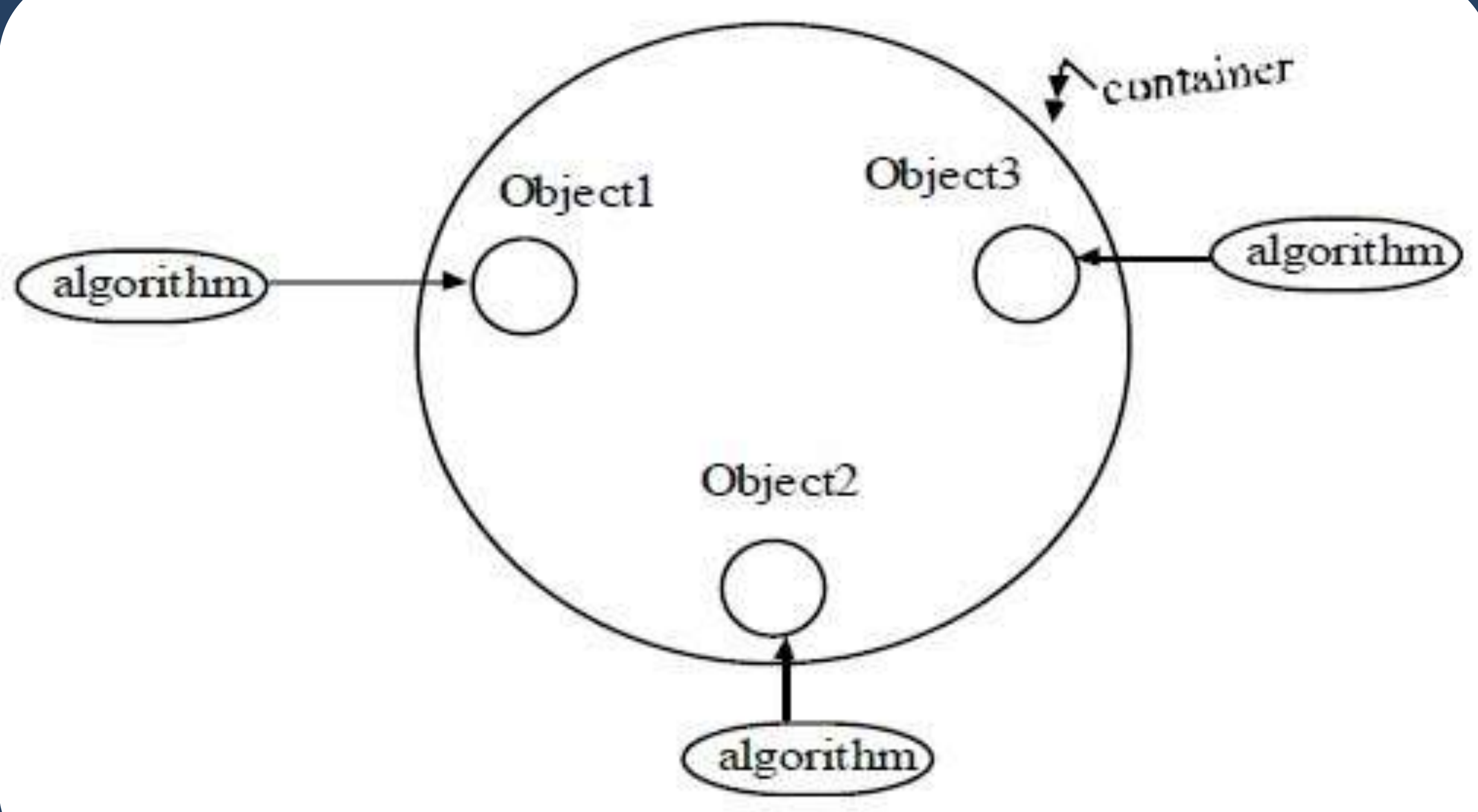
a. A template that defines a function with a generic type or types.
b. A template that defines a function with a specific type.
c. A template that defines a function with a specific number of parameters
d. None

INTRODUCTION TO STANDARD TEMPLATE LIBRARY

We learned about generic programming by using template class and template function in the early section. During the standardization of C++, Standard Template Library(STL) was included. It provides general purpose, templatized classes and functions that implement many popular and commonly used algorithms and data structures, for example, vector, stack queue, list map, etc. As the STL is constructed from template classes, the algorithm and data structure can be applied to any type of data.

STL is large and complex and it is difficult to discuss all the features. STL components that are now part of the Standard C++ library are defined in the namespace std. So we should use this directive to use STL.

The STL contains several components. But in core, it contains three fundamental components: containers, algorithms, and iterators. Their relationship is shown in the figure



INTRODUCTION TO STANDARD TEMPLATE LIBRARY

1. Containers

Containers are objects that hold other objects. It is a way in which data is organized in memory. The STL containers are implemented by template classes and therefore can be easily customized to hold different types of data. The STL defines ten containers that are grouped into three categories:

a) Sequence containers

The sequence container is a variable-sized container whose elements are arranged in a strict linear order.It supports insertion and removal of elements. Every data, user defined or built-in, has a specific position in the container. Sequence containers store elements in a linear list. Each element is related to one other element by its position along the line. They are decided into the following:

i) Vector
ii) List
iii) Dequeue

INTRODUCTION TO STANDARD TEMPLATE LIBRARY

b) Associative Containers

An associative container is a variable-sized container that supports efficient retrieval of elements based on keys. Like sequence container, it supports insertion and removal of elements, but differs from a sequence in that it does not provide a mechanism for inserting an element at a specific position. They are not sequential. There are four types of associative containers:

i) Set
ii) Multiset
iii) Map
iv) Multimap

c) Derived Containers

These containers are derived from sequential container. These are also known as container adaptors. The STL provides three derived containers:

i) Stack
ii) Queue
iii) Priority_queue

INTRODUCTION TO STANDARD TEMPLATE LIBRARY

2. Algorithms

An algorithm is a procedure that is used to process the data contained in the containers. STL provides more than sixty standard algorithms to support more extended or complex operations. Standard algorithms also permit us to work with two different types of containers at the same time. STL algorithms are not member functions or friends of containers. They are standalone template functions. STL algorithm based on the nature of operations they perform may be categorized as:

a) Mutating algorithms
b) b) Sorting algorithms
c) Set algorithms
d) Relational algorithms
e) Retrieve or non mutating algorithms

INTRODUCTION TO STANDARD TEMPLATE LIBRARY

3. Iterators

Iterators behave like pointers and are used to access individual elements in containers.

They are often used to traverse from one element to another, a process known as iterating through the container.

That means if the iterator points to one element in the range then it is possible to increase or decrease the iterator so that we can access the next element in the range.

Iterators connect algorithms and play a key role in the manipulation of data stored in the containers.

There are five types of iterators: input, output, forward, bidirectional and random.

1. What is a Standard Template Library in C++?

a. A library of generic algorithms, containers and iterators.
b. A library of pre-written code for common programming tasks.
c. A library of advanced programming techniques in C++.
d. None

2. Which of the following is a container class in the STL?

a. Sort
b. vector
c. transform
d. None

3. What is the syntax for creating a vector container in the STL?

a. MyVector<int> vector;
b. Vector myVector<int>;
c. Vector<int> myVector;
d. None

4. What is the purpose of an algorithm in the STL?

a. To create a container
b. To hold data in container
c. To perform a specific task on a container
d. None

3. What is the difference between a vector and an array in the STL?

a. A vector is a dynamic array while an array is an static array.
b. A vector can only hold primitive data types while an array can hold any data type.
c. A vector is faster than an array
d. None

4. What is an algorithm in the STL?

a. A data structure that stores objects of a particular type
b. A function that manipulates data stored in containers.
c. An interator that allows access to elements of a container
d. None

5. What is a function object in STL?

a. An object that stores data
b. An object that manipulates functions
c. An object that acts like a function
d. None

6. What is an iterator in the STL?

a. A function that returns the value of the last element in a container
b. A function that inserts an element at the end of a container.
c. A pointer to an element in a container
d. None

EXCEPTION HANDLING

ERROR HANDLING

The two most common types of bugs:

1. **Logic error:** Due to poor understanding of the problem and solution procedure.
2. **Syntactic error:** Due to poor understanding of language itself.

We often come across some peculiar problems other than logic and syntax errors. They are known as exceptions. Exceptions are run time anomalies or unusual conditions that a program may encounter while executing. E.g. Division by zero,access to an array outside its bound, running out of memory or disk space etc. The purpose of the exception handling mechanism is to provide means to detect and report “exceptional circumstances” so that appropriate action can be taken.

EXCEPTION HANDLING CONSTRUCTS (TRY, CATCH, THROW)

Exceptions are the anomalous or unusual events that change the normal flow of the program. Exception handling is the mechanism by which we can identify and deal with such unusual conditions. This 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 the exception, and the other to catch the exception and to take corrective actions.

CONSTRUCTS OF EXCEPTION HANDLING

In C++, the following keywords are used for exception handling.

a) try

b) catch

c) throw

1. 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.
2. A catch block defined by the keyword catch is used to catch the exception thrown by the throw statement in the try block, and handles it properly.

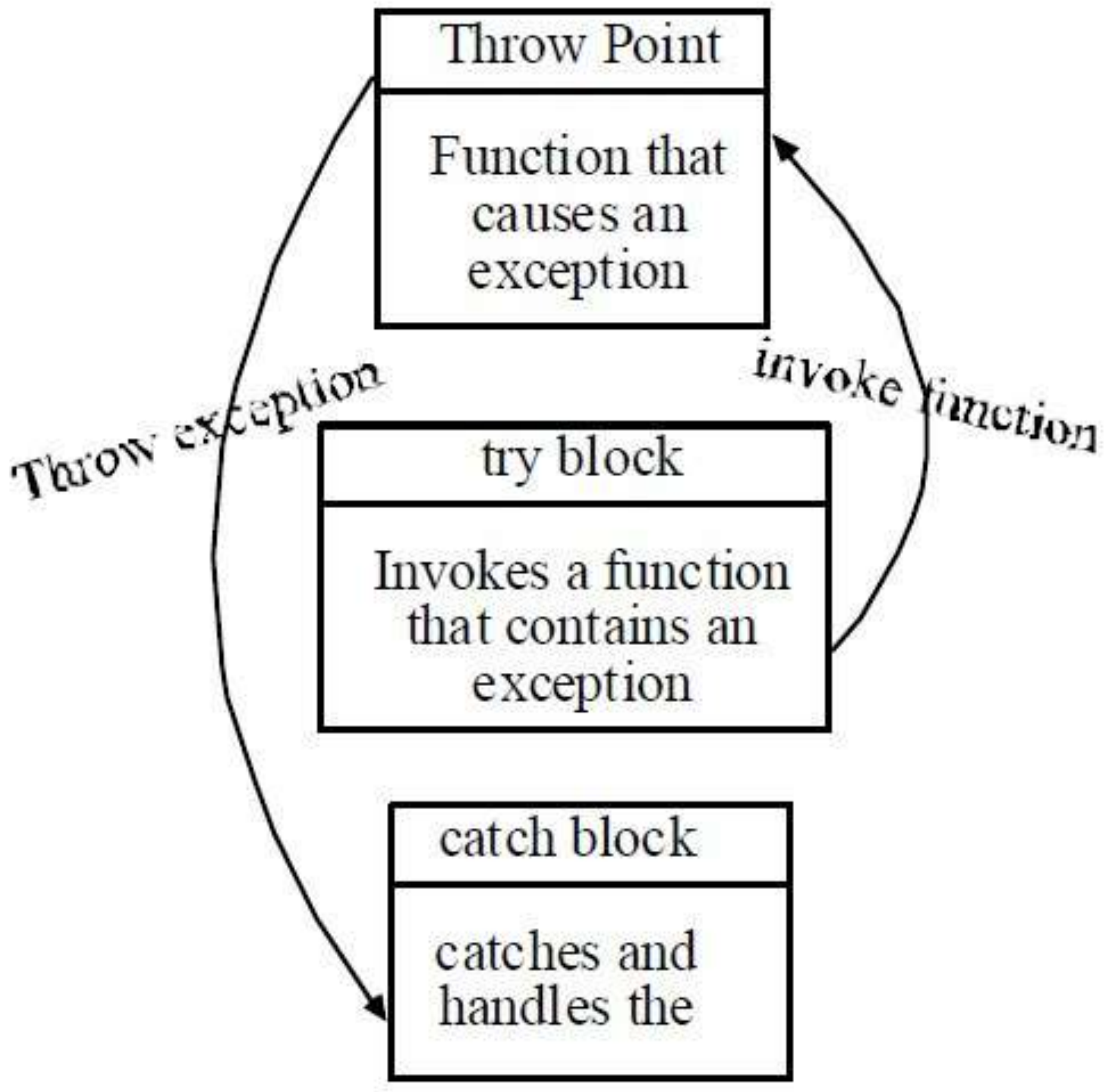


Fig : Function invoked by try block throwing exception

CONSTRUCTS OF EXCEPTION HANDLING

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

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

CONSTRUCTS OF EXCEPTION HANDLING

Try keyword is used to a block of statements surrounded by braces which may generate exceptions. When an exception is detected, it is thrown using the throw statement in try block, then the program control leaves the try block and enters the catch block comparing the catch argument try and the exception thrown type. If the type of the object matches the argument type in the catch statement, then the catch block is executed for handling the exception. If they don't 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 i.e. catch block is skipped. In handling exceptions we need to create a new kind of entity called an exception class.

CONSTRUCTS OF EXCEPTION HANDLING

```
#include<iostream>
using namespace std;
int main()
{
      int a,b,x;
      cout<<"Enter the values of a and b: ";
      cin>>a>>b;
      x = a-b;
      try
      {
       if(x!=0)
             cout<<"The result of (a/x) is "<<a/x<<endl;
       else
             throw(x);
      }
      catch(int i)
      {
             cout<<"Division by zero. X = "<<i<<endl;
      }
      return 0;
}
```

Output:

Enter the values of a and b: 4 2

The result of (a/x) is 2.

Output 2:

Enter the values of a and b: 8 8

Division by zero. X = 0

ADVANTAGE OVER CONVENTIONAL ERROR HANDLING

The error handling mechanism is somewhat new and very convenient in case of object oriented approach over conventional programming. When an error is detected, the error could be handled locally or not locally. Traditionally, when the error is not handled locally the function could

1. Terminate the program.
2. Return a value that indicates error.
3. Return some value and set the program in an illegal state.

The exception handling mechanism provides alternatives to these traditional techniques when they are dirty, insufficient and error prone. However, in the absence of exception all these three traditional techniques are used. Exception handling separates the error handling code from the other code making the program more readable. If the exception is not handled then the program terminates. Exception provides a way for code that detects a problem from which it cannot recover to the part of the code that might take necessary measures.

MULTIPLE EXCEPTION HANDLING

It is possible that a program segment has more than one condition to throw an exception. In such cases, we can associate more than one catch statement with a try statement(much like the conditions in a switch statement).

```
try
{
  //try block
}
catch(type1 arg)
{
 //catch block1
}
...........
...........
catch(typeN  arg)
{
 //catch blockN
}
```

MULTIPLE EXCEPTION HANDLING

```
#include<iostream>
using namespace std;
void test(int a, int b)
{
 try
     {
         if (b < 0)
                  throw b;
         if(b == 0)
                  throw 1.1;
         cout << "The Quotient = " << (a/b) << endl;
         cout<<"End of try block"<<endl;
    }
catch(int c)
{
 cout << "\nSecond Operand is less than zero" << endl;
 cout<<"caught an integer"<<endl;
}
catch(double c)
{
cout<<"\nSecond operand is equal to zero"<<endl;
cout<<"caught a double"<<endl;
}
cout<<"End of try-catch system"<<endl;
}

int main()
{
test(12,3);
test(12, -3);
test(12,0);
return 0;
}
```

Output:

```
The Quotient = 4
End of try block
End of try-catch system
Second Operand is less than zero
caught an integer
End of try-catch system
Second operand is equal to zero
caught a double
End of try-catch system
```

RE-THROWING EXCEPTION

We can make the handler(catch block) to rethrow the exception caught without processing it. In such situations, we may simply invoke “throw” without any argument.

throw;

This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement of that try/catch block.

```
#include<iostream>
using namespace std;
void Calculate(double a,double b)
{
    cout<<”Inside the function”<<endl;
    try
        {
        if(b==0)
            throw b; //throwing double
        else
            cout<<"\nThe result is:"<<a/b<<endl;
    }
    catch(double)    //catch a double
    {
        cout<<"\nCaught double value inside the catch block of the function";
        throw;     //rethrowing double
    }
}
```

```
int main()
{
    cout<<”Inside the main block”<<endl;
    try
    {
        calculate(10.5,2.0);
        calculate(10.5,0.0);
    }
    catch(double)
    {
        cout<<"\nInside the Catch block if the main() function";
        cout<<"\nException due to 0"<<endl;
    }
    return 0;
    }
```

RE-THROWING EXCEPTION

Output:

Inside the main block
Inside the function
The result is 5.25

Inside the function
Caught double value inside the catch block of the function
Inside the Catch block if the main() function
Exception due to 0

CATCHING ALL EXCEPTION

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
}
```

CATCHING ALL EXCEPTION

```
#include<iostream>
using namespace std;
void test(int x)
{
      try
      {
            if(x   ==   0)
            throw      x;
            if(x   ==   -1)
            throw    'x';
            if(x   ==   1)
            throw 1.0;
      }
      catch(...)    //catch all
      {
             cout<<"caught an exception";
      }
}
```

```
int main()
{
       cout<<"Testing generic catch"<<endl;
      test(-1);
      test(0);
      test(1);
      return 0;
}
```

Output:

```
Testing generic catch
caught  an  exception
caught  an  exception
caught an exception
```

NOTE: It may be a good idea to use the catch(...) as a default statement along with other catch handlers so that it can catch all those exceptions which are not handled explicitly. Remember, catch(..) should always be placed last in the list of catch handlers. Placing it before other catch blocks would prevent those blocks from catching exceptions.

EXCEPTION WITH ARGUMENTS

There may be situations where we need more information about the cause of exception. Let us consider there may be more than one function that throws the same exception in a program. It would be nice if we know which function threw the exception and the cause to throw an exception. This can be accomplished by defining the object in the exception handler. This means, adding data members to the exception class which can be retrieved by the exception handler to know the cause. While throwing, a throw statement throws an exception class object with some initial value, which is used by the exception handler.

EXCEPTION WITH ARGUMENTS

```
#include<iostream>
using namespace std;
class argument  {

       int a, b;
       public:
              string    msg;
              argument() {
              a= b= 0;
              }
               argument(string name)
               {
                     msg = name;
              }
               void input()
               {
                     cout << "Enter the value of a and b : ";
                     cin >> a >> b;
              }
              void calculate()
              {
                     if(b == 0)
                     throw argument("Divided by Zero.");
                     if(b < 0)
                     throw argument("Divided by Negative number");
                    cout << "Quotient = " << a/b << endl;
              }
       };

       int main()
       {
              argument A1;
              A1.input();
              try {
                     A1.calculate();
               }
               catch(argument A)
               {
                     cout << "Exception : " << A.msg << endl;
              }
               return 0;
        }
```

EXCEPTION SPECIFICATION FOR FUNCTION

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:

```
return_type function_name(arg_list) throw(type list)
{
 //body of function
}
```

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 ro prevent a function from throwing any exception, we may do so bymaking the type list empty i.e.

```
return_type function_name(arg_list) throw()
{
//function body
}
```

EXCEPTION SPECIFICATION FOR FUNCTION

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

void calculate(int a,int b) throw(int)
{
    if(b==0)
        throw 2.0;
    if(b<0)
        throw 1;
    cout<<"\nQuotation is :"<<a/b;
}
```

```
int main()
{
    int c,d;
    cout<<"\nEnter the value of c and d:";
    cin>>c>>d;
    try {
        calculate(c,d);
    }
    catch(double)
    {
        cout<<"\nException as Second operand is -ve";
    }

    return 0;
}
```

Output1:

Enter the value of c and d:
5
0

Output2:

Enter the value of c and d:
5
-2
Exception as Second operand is -ve

In first run, the calculate function is throwing double value which is not specified in the throw list. Hence the program terminates. In second run, the calculate function is throwing an int exception, but there is no catch handler for int. So the program terminates.

HANDLING UNCAUGHT AND UNEXPECTED EXCEPTIONS

Handling uncaught exceptions:

If the exception is thrown and no exception handler is found(i.e. the exception is not caught) - the program calls the terminate() function. We can specify our own termination function with the set_terminate() function.

```
#include<iostream>
using namespace std;
void test_handler()
{
      cout<<"Program is terminated....";
}
```

```
int main()
{
      set_terminate(test_handler);
      try
      {
            cout<<"Inside try block."<<endl;
            throw 10;
      }
      catch(char c)
      {
            cout<<"caught exception";
      }

      return 0;
}
```

Output:
Inside try block Program is terminated....

Here, int is thrown but there is no catch handler to catch the int exception. So, the program is terminates calling the test_handler() function.

HANDLING UNCAUGHT AND UNEXPECTED EXCEPTIONS

Handling unexpected exceptions:

Similar to uncatch exception, When a function attempts to throw an exception that is not in the throw list, unexpected() function is invoked. Like in terminate() function, we can specify the function that is called by the unexpected() with the help of set_unexpected() function.

```
#include<iostream>
using namespace std;
void test_unexpected()
{
     cout<<”Program terminated due to unexpected exception”<<endl;
}
void calculate( int a, int b) throw(int)
{
     if(b == 0)
          throw ‘A’;
     if(b<0)
          throw 1.0; //double value is thrown which is not allowed
     cout<<”Result is :”<<a/b<<endl;
}
```

```
int main()
{
     set_unexpected(test_unexpected);
     try
     {
          calculate(2,-7);
     }
     catch(int c)
     {
          cout<<”Integer Exception”<<endl;
     }
     catch(double d)
     {
          cout<<”Double Exception”<<endl;
     }
     return 0;
}
```

Output:

Program terminated due to unexpected exception

Here, calculate() function is trying to throw a double value which is not included in the throw list, So, an unexpected exception occurs which calls the test_unexpected().

THANK YOU