Function Templates and Exception Handling

Templates

An important feature of C++ is templates which provide great flexibility to the language. Templates support generic programming, allows developing reusable software components such as function, classes, etc supporting different data types in a single framework. For example, function such as sort, swap which support various data types can be developed.

A template in C++ allows the construction of a family of template functions and class to perform the same operation on different data types. The templates declared for function are called function templates and those declared for classes are called class templates. They perform appropriate operations depending on the data type of the parameters passed to them. It allows a single template to deal with a generic data type T.

Function Templates

The limitation of function is they can operate only on a particular data type. It can be overcome by defining that function as a function template or generic function. A function template specifies how an individual function can be constructed. Consider the following program:

```
//function overloading
//multiple function with same name
//showing need for template

#include<iostream.h>
#include<conio.h>
int max(int ,int);
long max(long, long);
float max(float,float);
char max(char,char);
void main()
{
    int i1=15,i2=20;
    cout<<"Greater is "<<max(i1,i2)<<endl;
    long l1=40000, l2=38000;
    cout<<"Greater is "<<max(l1,l2)<<endl;
    float f1=55.05, f2=67.777;
    cout<<"Greater is "<<max(f1,f2)<<endl;
    char c1='a',c2='A';
    cout<<"Greater is "<<max(c1,c2)<<endl;
    getch();
}

int max(int i1, int i2)
{
      return(i1>i2?i1:i2);
}

long max(long l1, long l2)
{
      return(l1>l2?l1:l2);
}
```

```
float max(float f1, float f2)
{
      return(f1>f2?f1:f2);
}

char max(char c1, char c2)
{
      return(c1>c2?c1:c2);
}
```

Above program of multiple max functions are used to find greater value among two, for different data types. This illustrates the need for function templates. The program consists of 4 max functions.

```
      int max(int ,int);
      long max(long, long);
      float max(float,float);
      char max(char,char);
```

Whose logic of finding greater value is same and differs only in terms of data types. The C++ template features enables substitution of a single piece of code for all these overloaded function as follows.

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

Such functions are known as function templates. When max operation is requested on operands of any data types, the compiler creates a function internally without the user intervention and invokes the same.

A function template is prefixed with the keyword template and a list if template type arguments. The template-type arguments are called generic data types, since their memory requirement and data representation is not known in the declaration of the function template. It is known only at the point of a call to a function template.

```
Template <class T, …..>
Return_type function_name(arguments)
{
      ……..//body of template
      ……..
}
```

```
//find greater using template
#include<iostream.h>
#include<conio.h>

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

```
void main()
{
    int i1=15,i2=20;
    cout<<"Greater is "<<max(i1,i2)<<endl;
    long l1=40000, l2=38000;
    cout<<"Greater is "<<max(l1,l2)<<endl;
    float f1=55.05, f2=67.777;
    cout<<"Greater is "<<max(f1,f2)<<endl;
    char c1='a',c2='A';
    cout<<"Greater is "<<max(c1,c2)<<endl;
    getch();
}
```

Function and Function Template

Function templates are not suitable for handling all data types, so it is necessary to override function templates by using normal function for specific data types. If a program has both the function and function template with the same name, first compiler selects the normal function, if it matches with the requested data type, otherwise it creates a function using a function template.

```
//function with function template
#include<iostream.h>
#include<string.h>
#include<conio.h>

template <class T>
T max( T a, T b)
{
      return(a>b?a:b);
}

//for string data types
char *max(char *a, char *b)
{
      if(strcmp(a,b)>0)
      return a;
   else
      return b;
}

void main()
{
    int i1=15,i2=20;
    cout<<"Greater is "<<max(i1,i2)<<endl;
    long l1=40000, l2=38000;
    cout<<"Greater is "<<max(l1,l2)<<endl;
    float f1=55.05, f2=67.777;
    cout<<"Greater is "<<max(f1,f2)<<endl;
    char c1='a',c2='A';
    cout<<"Greater is "<<max(c1,c2)<<endl;
    char str1[]="apple", str2[]="zebra";
    cout<<"greater is "<<max(str1,str2);
    getch();
}
```

Overloaded Function Templates

The function templates can also be overloaded with multiple declarations. Similar to overloading of normal functions, overloaded function templates must differ either in terms of number of parameters or their type.

```
//overloading function template
//overloaded function templates
#include<iostream.h>
#include<conio.h>

template <class T>
void print(T data)
{
      cout<<data<<endl;
}
template <class T>
void print(T data, int n)
{
      for(int i=0;i<n;i++)
      cout<<data<<endl;
}

void main()
{
      print(1);  // 1
   print(1.5);    //1.5
   print(420,2);       //420 two times
   print("my Nepal my pride",3); //3 times
   getch();
}
```

Class Templates

Similar to functions, classes can also be declared to operate on different data types. Such classes are called class templates. A class template specifies how individual classes can be constructed similar to normal class specification. These classes model a generic class which supports similar operations for different data types. A generic stack like generic function can be created which can be used for storing data of type integer, floating number, character etc.

```
//implementation of stack class as template
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>
#define max 20

template <class T>
class stack
{
      private:
      T s[max];
      int top;
   public:
```

```
        stack()   //constructor
        { top=-1;}

        void push(T x)//put number on stack
        {

                    s[++top]=x;
        }
        T pop()//take number from stack
        {

             return s[top--];
        }
};

void main()
{
      //for integer data type
      stack <int> s1;

      s1.push(11);
      s1.push(22);
      s1.push(33);
      cout<<"\nNumber Popped:"<<s1.pop(); //33
      cout<<"\nNumber Popped:"<<s1.pop(); //22
      s1.push(44);
      cout<<"\nNumber Popped:"<<s1.pop(); //44
      //for floating point data type
      stack <float> s2;
      s2.push(11.11);
      s2.push(22.22);
      s2.push(33.33);
      cout<<"\nNumber Popped:"<<s2.pop(); //33.33
      cout<<"\nNumber Popped:"<<s2.pop(); //22.22
      s2.push(44.44);
      cout<<"\nNumber Popped:"<<s2.pop(); //44.44

      //for character data type
      stack <char> s3;
      s3.push('A');
      s3.push('B');
      s3.push('C');
      cout<<"\nNumber Popped:"<<s3.pop(); //C
      cout<<"\nCharcter Popped:"<<s3.pop(); //B
      s3.push('D');
      cout<<"\nCharcter Popped:"<<s3.pop(); //D
      getch();
}
```

Exception Handling

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

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

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

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called *handlers*.

To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a *try-block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

An exception is thrown by using the `throw` keyword from inside the `try` block. Exception handlers are declared with the keyword `catch`, which must be placed immediately after the `try` block:

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

int main () {
  try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "An exception occurred.
Exception Nr. " << e << '\n';
  }
  return 0;
}
```

```
An exception occurred.
Exception Nr. 20
```

Edit & Run

The code under exception handling is enclosed in a `try` block. In this example this code simply throws an exception:

```
throw 20;
```

A `throw` expression accepts one parameter (in this case the integer value `20`), which is passed as an argument to the exception handler.

The exception handler is declared with the `catch` keyword immediately after the closing brace of the `try` block. The syntax for `catch` is similar to a regular function with one parameter. The type of this parameter is very important, since the type of the argument passed by the `throw` expression is checked against it, and only in the case they match, the exception is caught by that handler.

Multiple handlers (i.e., `catch` expressions) can be chained; each one with a different parameter type. Only the handler whose argument type matches the type of the exception specified in the `throw` statement is executed.

If an ellipsis (`...`) is used as the parameter of `catch`, that handler will catch any exception no matter what the type of the exception thrown. This can be used as a default handler that catches all exceptions not caught by other handlers:

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

In this case, the last handler would catch any exception thrown of a type that is neither `int` nor `char`.

After an exception has been handled the program, execution resumes after the *try-catch* block, not after the `throw`statement!.

It is also possible to nest `try-catch` blocks within more external `try` blocks. In these cases, we have the possibility that an internal `catch` block forwards the exception to its external level. This is done with the expression `throw;` with no arguments. For example:

```
try {
  try {
      // code here
  }
  catch (int n) {
      throw;
  }
}
```

```
9  catch (...) {
10   cout << "Exception occurred";
11 }
```

Exception specification

Older code may contain *dynamic exception specifications*. They are now deprecated in C++, but still supported. A *dynamic exception specification* follows the declaration of a function, appending a `throw` specifier to it. For example:

```
double myfunction (char param) throw (int);
```

This declares a function called `myfunction`, which takes one argument of type `char` and returns a value of type `double`. If this function throws an exception of some type other than `int`, the function calls std::unexpected instead of looking for a handler or calling std::terminate.

If this `throw` specifier is left empty with no type, this means that std::unexpected is called for any exception. Functions with no `throw` specifier (regular functions) never call std::unexpected, but follow the normal path of looking for their exception handler.

```
1 int myfunction (int param) throw(); // all exceptions call unexpected
2 int myfunction (int param);         // normal exception handling
```

```
/* Program for Exception Handling Divide by zero Using
C++ Programming */

#include<iostream.h>

#include<conio.h>
void main()
{
int a,b,c;
float d;
clrscr();
cout<<"Enter the value of a:";
cin>>a;
cout<<"Enter the value of b:";
cin>>b;
cout<<"Enter the value of c:";
cin>>c;

try
{
if((a-b)!=0)
{
d=c/(a-b);
cout<<"Result is:"<<d;
}
else
{
throw(a-b);
}
}
catch(int i)
{
cout<<"Answer is infinite because a-b is:"<<i;
}

getch();
}
```

Output

Enter the value for a: 20
Enter the value for b: 20
Enter the value for c: 40

Answer is infinite because a-b is: 0