Lab Sheet 9
Understanding the Concept of Templates and Exception

Function Template

Overloaded functions normally are used to perform similar operations on different types of data. If the operations are identical for each type, it is more convenient to use function templates. The programmer writes a single function-template definition. Based on the argument types provided explicitly or inferred from calls to this function, the compiler generates separate object-code function to handle each function call appropriately.

All function-template definition begin with keyword **template** followed by a list of formal type parameters to the function template enclosed in angle brackets (< and >); each formal type parameter must be preceded by either of the interchangeable keywords **class** or **typename**, as in

```
template< class Type >
```

Or

```
template< typename ElementType >
```

Or

```
template< class Type_1, class Type_2>
```

Example

```cpp
#include<iostream>

using namespace std;



//function template printArray
definition

template< class T>

void printArray(T *array,int count)

{

for(int i=0;i<cout;i++)

cout<<array[i]<<" "<<endl;

}

//end of the function

int main()

{

int a[4]={1,2,3,4};

double b[5]={1.1,2.2,3.3,4.4,5.5};

char c[6]= "Hello";

cout<<" Array a contains: "<<endl;

printArray(a,4);

cout<<" Array b contains: "<<endl;

printArray(b,5);

cout<<" Array c contains: "<<endl;

printArray(c,6);

return 0;

}
```

Class Template

The template concept can be extended to classes. Class templates are generally used for data storage classes. The approach is similar to that used in function template. Here the **template** keyword and **class** name signal that the entire class will be a template.

```
template <class T>

Class Stack

{

//data and member functions using argument T

};
```

Example

```
#include<iostream>                    template<class T>

using namespace std;                  T Stack<T>::pop()

template<class T>                     {

class Stack                           return st[top--];

{                                     }

private:

T st[100];                            int main()

int top;                              {

public:                               Stack<float> s1;

Stack();                              s1.push(111.1F);

void push(T var);                     s1.push(222.2F);

T pop();                              s1.push(333.3F);

};                                    cout<<"1 : "<<s1.pop()<<endl;

                                      cout<<"2 : "<<s1.pop()<<endl;

template<class T>                     cout<<"3 : "<<s1.pop()<<endl;

void Stack<T>::Stack()

{                                     Stack<long> s2;

top=-1;                               s2.push(123123123L);
```

```
}                                       s2.push(234234234L);

template<class T>                       s2.push(345345345L);

void Stack<T>::push(T var)              cout<<"1 : "<<s2.pop()<<endl;

{                                       cout<<"2 : "<<s2.pop()<<endl;

st[++top]=var;                          cout<<"3 : "<<s2.pop()<<endl;

}                                       return 0;

                                        }
```

Exception Handling

Exception handling is designed to handle only synchronous exceptions. The mechanism provides 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)

General syntax

```
.........

try

{
        ..........
        throw exception object;    //Block of statements which detects and throws an exception
        .........

}

catch(type arg)                 //Catches exception

{

        .........                //Block of statements that handles the exception

}

.........
```

. . . .

Example

```cpp
#include<iostream>

using namespace std;

class ex_demo
{
  private:
    int a;
    int b;
  public:
    ex_demo(){}
    class DIVIDE{}; //abstract class for
exception
    void getdata()
    {
        cout<<"Enter dividend (A) and
        divisor (B)" <<endl;
        cin>>a>>b;
    }
    void divide()
    {
      try
      {
        if(b!=0)
        {
        cout<<"Result(A/B) =
"<<a/b<<endl;
        }
        else
        {
          throw DIVIDE();
        }
      }
      catch(DIVIDE)
      {
              cout<<"Exception Caught
        : b= " <<b<<endl;
      }
    }
};

int main()
{
  clrscr();
  ex demo ex;
  ex.getdata();
  ex.divide();
  return 0;
}
```

Multiple Catch Statement

A program can have more than one condition to throw an exception. It is possible to use multiple catch statements with try block. When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match is executed, after executing the handler; the control goes to the first statement after the last catch block for that try.

Syntax

```
try                                     catch(type2 arg)

{                                       {

  //try block                             //catch block

}                                       }

catch(type1 arg)                        ....

{                                       ....

  //catch block                         catch(typeN arg)

}                                       {

                                          //catch block

                                        }
```

Re-throwing an exception

A handler may decide to re-throw the exception caught without processing it. In such situation, we may simply invoke throw without any arguments as shown below:

```
throw;
```

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

Exercises

1. Create a function called sum **( )** that returns the sum of the elements of an array. Make this function into a template so it will work with any numerical type. Write a **main ( )** program that applies this function to data of various type.
2. Write a class template for queue class. Assume the programmer using the queue won't make mistakes, like exceeding the capacity of the queue, or trying to remove an item when the queue is empty. Define several queues of different data types and insert and remove data from them.
3. Modify the stack class given in the previous lab to add the exception when user tries to add item while the stack is full and when user tries to add item while the stack is empty. Throw exception in both of the cases and handle these exceptions.
4. Write any program that demonstrates the use of multiple catch handling, re-throwing an exception, and catching all exception.