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

```
#include<iostr eam>
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

```
void push(T var);

#include<iostr

eam> using

namespace std;

template<class

T> class Stack

{

private:

T st[100]

; int

top;

public:

Stack();
```

```
template<clas
     s T>

T Stack<T>::pop()

{

return st[top--];

}
```

```
 T pop();

 };
```

```
template<class
      T>
```

```
int main()

{

Stack<float> s1;

s1.push(111.1F);

s1.push(222.2F);
```

```
s1.push(333.3F);

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

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

cout<<"3 :
"<<s1.pop()<<endl;
```

```
void Stack<T>::Stack()

{

top=-1;
```

```
Stack<long> s2;

s2.push(123123123L);
```

```
}

template<class T>

void Stack<T>::push(T
var)

{

st[++top]=var;


}
```

```
s2.push(234234234L);

s2.push(345345345L);

cout<<"1 :
"<<s2.pop()<<endl;

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

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
```

```
{
    ..........          //Block of statements which
    throw
    exception_obj       detects and throws an exception
    ect;
    .........

}

catch(type arg)     //Catches exception

{

    .........       //Block of statements that handles

                    the exception
}

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

....

Example

```
#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;

        }
```

Multiple Catch Statement

```
        else
        {
         throw DIVIDE();
        }
       }
       catch(DIVIDE)
       {
              cout<<"Exce
          ption Caught :
          b= " <<b<<endl;
```

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

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
{
 //try block
}
catch(type1 arg)
{
 //catch block
}
catch(type2 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.

```
#include <iostream>
using namespace std;
template < typename T>
T sum(T array[],int n)
{
  T s= 0;
  for(int i = 0 ; i < n; i++)
  {
    s+=array[i];
  }
  return s;
}
int main()
{
  int num[] = {4,5,6};
  float fnum[] = {4.0,3.0,5.5};
  cout << sum(num,3) << endl;
```

```
  cout << sum(fnum,3) << endl;
  return 0;
}
```

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.

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

template <typename T>
class Queue
{
 private:
  T data[100];
  int pos;
 public:
  Queue()
  {
    pos = 0;
    for(int i = 0; i< 100; i++)
      data[i]=0;
  }
  void add(T d)
  {
    data[pos] = d;
    pos++;

  }
  T get()
  {
    T  d = data[0];
    for(int i = 0 ; i < pos; i++)
    {
      data[i] = data[i+1];

    }
    pos--;
    return d;
  }
```

```
};
int main()
{
  Queue<int> intlist;
  intlist.add(3);
  intlist.add(4);
  cout << intlist.get() << endl;
  cout << intlist.get() << endl;
  Queue<float> flist;
  flist.add(3.0);
  flist.add(4.9);
  cout << flist.get() << endl;
  cout << flist.get() << endl;
  return 0;
}
```

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.

```
#include<iostream>
#include<cstring>
using namespace std;
#define SIZE 2
class exc{
 public:
  string info;
};
template<class T>
class Stack
{
 private:
  T st[SIZE];
  int top;
 public:
  Stack();
  void push(T var);
  T pop();
};

template<class T>
Stack<T>::Stack()
{
```

```
  top=-1;
}

template<class T>
void Stack<T>::push(T var)
{
  try
  {
    if(top >= (SIZE-1))
    {
      top = SIZE - 1;
      exc  e;
      e.info="Stack is full";
      throw  e;
    }
    else
    {
      st[++top]=var;
    }
  }
  catch(exc e)
  {
    cerr << e.info << endl;
  }
}

template<class t>
t Stack<t>::pop()
{
  try {
    if (top < 0)
    {
      exc e;
      e.info ="stack is empty";
      throw e;
    }
    else
    {
      return st[top--];
    }
  }
  catch(exc e)
  {
    cerr << e.info << endl;
  }
}
```

```
int main()
{
  Stack<float> s1;

  s1.push(111.1F);
  s1.push(222.2F);
  s1.push(333.3F);

  cout<<"1 : "<<s1.pop()<<endl;
  cout<<"2 : "<<s1.pop()<<endl;
  cout<<"3 : "<<s1.pop()<<endl;

  Stack<long> s2;

  s2.push(123123123L);
  s2.push(234234234L);
  s2.push(345345345L);

  cout<<"1 : "<<s2.pop()<<endl;
  cout<<"2 : "<<s2.pop()<<endl;
  cout<<"3 : "<<s2.pop()<<endl;

  return 0;
}
```

4. Write any program that demonstrates the use of multiple catch handling, re-throwing an exception, and catching all exception.

```
#include <iostream>
using namespace std;
class DIVZERO{};
class DIVMINUS{};
int main()
{
  int a, b;
  float ans;
  try {
    cout << "a";
    cin >> a;
    cout << "b";
```

```
    cin >> b;

    try {
      if(b < 0)
        throw DIVMINUS();
      if(b == 0)
        throw DIVZERO();
      ans = a/b;
    }
    catch (DIVZERO)
    {
      cerr << "rethrowing DIVZERO exception" <<
endl;
      throw;
    }
    catch (DIVMINUS)
    {
      cerr << "divison by minus in not allowed"<<
endl;
    }

  } catch (...) {
    cerr << "caught exception";
  }
  cout << ans;
  return 0;
}
```