Unit 7
Function Templates and Exception Handling

```cpp
//program showing the neeed for template
#include<iostream>
using namespace std;
int max(int a, int b)
{
    int max;
    if(a>b)
        max=a;
    else
        max=b;
    return max;
}
float max(float a, float b)
{
    float max;
    if(a>b)
        max=a;
    else
        max=b;
    return max;
}
double max(double a, double b)
{
    double max;
    if(a>b)
        max=a;
    else
        max=b;
    return max;
}
char max(char a, char b)
{
    int max;
    if(a>b)
        max=a;
    else
        max=b;
    return max;
}
int main()
{
    int x=15, y= 20;
    cout<<"Greater is:"<<max(x,y)<<endl;
    float f1=200.34, f2= 2000.234;
    cout<<"Greater is:"<<max(f1,f2)<<endl;
    int d1=15.55, d2= 20.45;
    cout<<"Greater is:"<<max(d1,d2)<<endl;
    int c1='a', c2= 'A';
    cout<<"Greater is:"<<max(c1,c2)<<endl;
    return 0;
}
```

```
Greater is:20
Greater is:2000.23
Greater is:20.45
Greater is:97
```

Contd..

- In general if the same operation is to be performed with different data types then we need to write the same code(function or classes) for different data types.
- This causes redundant code in the program with only difference in data type.
  - For example, if we need to write functions for max finding (function returns the greater of two values) that support different data types then need to make functions for all different data types.
- But Templates features of C++ make it possible to use one function or class to handle many different data types.
- The previous program illustrates the need of templates

What is Template?

- An important feature of c++ which helps to eliminate redundant code by writing a single generic code that takes data type as argument.

- 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.

- The template functions are also called generic functions and template class are also called generic classes because they support any data type.

Contd..

- **Function Template:**
  - A function template can be used to operate on different types of data by taking data types as a parameter.
  - Template functions are defined by using the keyword template.

syntax for template definition:

```
template<class type1,.....>          //template declaration

type1 function_name(parameter list) //template function definition
{
    //function body
}
```

here class->used to specify generic data type type1.

This generic data type is used within function body to declare data.

Example:

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

- **How templates work?**
  - Templates are expanded at compiler time. The idea is simple, source code contains only generic function/class, but compiled code may contain multiple copies of same function/class.

```
#include <iostream>
using namespace std;
template<typename T> T myMax(T x , T y){
    return (x>y)?x:y;
}
int main() {
   cout<<myMax<int>(3,4)<<endl;
   cout<<myMax<float>(3.4,1.2)<<endl;
   cout<<myMax<char>('M','P')<<endl;

    return 0;
}
```

```
//program to find greater using template
#include<iostream>
using namespace std;
template<class T>
T Fmax(T a, T b)
{
    T max;
    if(a>b)
        max=a;
    else
        max=b;
    return max;
}
int main()
{
    int x=15, y= 20;
    cout<<"Greater is:"<<Fmax(x, y)<<endl;
    float f1=200.34, f2= 2000.234;
    cout<<"Greater is:"<<Fmax(f1,f2)<<endl;
    double d1=15.55, d2= 20.45;
    cout<<"Greater is:"<<Fmax(d1,d2)<<endl;
    int c1='a', c2= 'A';
    cout<<"Greater is:"<<Fmax(c1,c2)<<endl;
    return 0;
}
```

```
Greater is:20
Greater is:2000.23
Greater is:20.45
Greater is:97
```

Contd..

- **Homework:** Find minimum of two values by using template function

```
//program to find smaller using template
#include<iostream>
using namespace std;
template<typename T>
T Fmin(T a, T b)
{
     T min;
     if(a<b)
          min=a;
     else
          min=b;
     return min;
}
int main()
{
     int x=15, y= 20;
     cout<<"smaller is:"<<Fmin(x, y)<<endl;
     float f1=200.34, f2= 2000.234;
     cout<<"smaller is:"<<Fmin(f1,f2)<<endl;
     double d1=15.55, d2= 20.45;
     cout<<"smaller is:"<<Fmin(d1,d2)<<endl;
     int c1='a', c2= 'A';
     cout<<"smaller is:"<<Fmin(c1,c2)<<endl;
     return 0;
}
```

```
smaller is:15
smaller is:200.34
smaller is:15.55
smaller is:65
```

Contd..

- **Function template with multiple arguments:**
  - Can be define by specifying more template parameters between the angle brackets in function template declaration.
  - For example:

```
template<class T, class U>
T Fmin(T a, U b)
{
    T min;
    if(a<b)
        min=a;
    else
        min=b;
    return min;
}
```

  - In this case, our function template Fmin() accepts two parameters of different types and returns value of the same type as the first parameter (T) that is passed.
  - For example, after that declaration and definition we could call Fmin() with:

```
y= Fmin(x, z);
```

Contd..

```
//program to find minimum of two values of
//different types by using function templete
#include<iostream>
using namespace std;
template<class T, class U>
T Fmin(T a, U b)
{
    T min;
    if(a<b)
        min=a;
    else
        min=b;
    return min;
}
int main()
{
    int x=85, y;
    char z= 'a';
    y= Fmin(x, z);
    cout<<"smaller is:"<<y<<endl;
    return 0;
}
```

```
smaller is:85
```

Class Template

- The class template models a generic class which supports similar operation for different data types. The general form of defining class template is as follows:

```
syntax:
    template<class template_type,.......>
    class class_name
    {
        private:
            //data member of template type or non template type
        public:
            //function members with template type argument and return type
    };
```

Contd..

- The code `template<class template_type,.....>` indicates that a template is being declared.
- The keyword template before the class indicates that a class template is being declared.
- The keyword class inside angle brackets specifies a generic data type(template_type in this case).
- The identifier template_type is a template parameter that is used as a generic data type within class. There can be more than one template parameters with comma separated list with each template type preceded with keyword class.
- The template parameters are replaced by actual data type when the specific version of the class is created.

Contd..

- Once a class template is declared, we create a specific instance of the class using the following syntax:

```
class_name<data_type> object;
```

- Where data_type is the type specified as argument to class template. The data type specified here is used by the compiler to replace the template type parameter in the class template definition to create the specific version of the class.
- If int is used as data type then all the occurrence of the template parameter is replaced by int.

Contd..

- If the class template uses multiple template parameters then specific instance of class is created by passing multiple type arguments as required by the class template declared as:

```
class_name<data_type1, data_type2, ......>object;
```

- The process of generating a class definition from a class template is called template instantiation.
- A version of template for a particular template argument is called a specialization.

Contd..

- Difference between class template and function template:
  - A class template differ from function template in the way they are instantiated. Calling the function using specific argument type creates a function but to instantiate a class, we must pass the template arguments during object declaration.

- A class generated from a class template is a perfectly ordinary class. The class template does not add any runtime overhead and it does not reduce amount of code generated but it reduce the redundant code while writing the programs.

Contd..

```
//class template to find larger of two numbers
#include<iostream>
using namespace std;
template<class T>
class mypair
{
    T a,b;
    public:
        mypair(T first, T second)
        {
            a=first;
            b= second;
        }
        T getmax()
        {
            T retval;
            retval= a>b?a:b;
            return retval;
        }
};
int main()
{
    mypair <int>o1(100,75);
    cout<<"Larger= "<<o1.getmax()<<endl;
    mypair  <double>o2(90.9, 80.8);
    cout<<"Larger= "<<o2.getmax()<<endl;
    return 0;
}
```

```
Larger= 100
Larger= 90.9
```

```cpp
#include <iostream>
using namespace std;
template<typename T, typename S> class test{
    T a;
    S b;
    public:
        test(T f , S s){
            a = f;
            b =s;
        }
        S getmax(){
            S retval = a>b?a:b;
            return retval;
        }
};
int main() {
    test o1(199,200.5);
    cout<<o1.getmax()<<endl;
    test o2(68,'C');
    cout<<o2.getmax()<<endl;
     return 0;
}
```

```
/tmp/VNh8pfDtiJ.o
200.5
D
```

```cpp
#include <iostream>
using namespace std;
template<typename T, typename U>
class num{
        T a;
        U b;
    public:
        num(T n1, U n2){
            a = n1;
            b = n2;
        }
        T max(){
           T retval;
           if (a>b){retval =a;}
           else{retval = b;}
           return retval;
        }
};
```

```cpp
int main() {
    num <int,char>obj1(20,'A');
    cout<<"Larger is: "<<obj1.max();
    return 0;
}
```

```
/tmp/Dj3sytMrTU.o
Larger is: A

=== Code Execution Successful ===
```

Templates and inheritance

- Similar to inheritance of normal class, the class template can also be inheritated.
- When creating derived class with template mechanism we can create derived class with following ways:
  - Creating a non-template derived class from a template base class
  - Creating a template derived class without the template parameter from the base class template.
  - Creating a template derived class with template parameter from the base class
  - Creating a derived class with extra template parameter in the derived class along with the base class template parameter.
  - Creating a template derived class from non-template base class.

Contd..

Creating a non-template derived class from a template base class:

```
// Creating a non-template derived
//class from a template base class
#include<iostream>
using namespace std;
template<class T>
class base
{
    private:
        T data;
    public:
        base()
        {

        }
        base(T a)
        {
            data=a;
        }
        void display()
        {
            cout<<"data="<<data<<endl;
        }
};
class derived1:public base<int> // suppling template argument of base class
{   // this process replaces the template parameter of the base class and
    // eliminates the template parameter of the base class in the derived class

    public:
        derived1()
        {
        }
        derived1(int a): base<int>(a)
        {
        }
};
int main()
{
    derived1 obj(5);
    obj.display();
    return 0;
}
```

Output:

```
data=5
```

Contd..: Creating a template derived class without the template parameter from the base class template.

```
//Creating a template derived class without the
//template parameter from the base class template.
#include<iostream>
using namespace std;
template<class T>
class base
{
    private:
        T data;
    public:
        base()
        {

        }
        base(T a)
        {
            data=a;
        }
        void display()
        {
            cout<<"data="<<data<<endl;
        }
};
```

```
template<class T>   //add extra template parameter in the derived class
class derived1:public base<int> // suppling template argument of base class
{   // this process replaces the template parameter of the base class and
    // eliminates the template parameter of the base class in the derived cla
    private:
        T data;
    public:
        derived1()
        {
        }
        derived1(int a, T b): base<int>(a)
        {
            data=b;
        }
        void display()
        {
            cout<<"In base:";
            base<int>::display();
            cout<<"In derived class:"<<data<<endl;
        }
};
int main()
{
    derived1 <float> obj(10, 12.34);
    obj.display();
    return 0;
}
```

Output:

```
In base:data=10
In derived class:12.34
```

Contd..:

Creating a template derived class with template parameter from the base class

```
// Creating a template derived class with
// template parameter from the base class
#include<iostream>
using namespace std;
template<class T>
class base
{
    private:
        T data;
    public:
        base()
        {

        }
        base(T a)
        {
            data=a;
        }
        void display()
        {
            cout<<"data="<<data<<endl;
        }
};
template<class T>
class derived1:public base<T>
{
    public:
        derived1()
        {
        }
        derived1(int a): base<T>(a)
        {

        }
};
int main()
{
    derived1 <int> obj(12);
    obj.display();
    return 0;
}
```

Output:

```
data=12
```

Exception Handling

- Exceptional Handling (Try, throw and catch)
- Multiple exceptions
- Exceptions with arguments

Contd..

- **What are exceptions?**
  - Exceptions are errors that occur at runtime.
  - They are caused by a wide variety of exceptional circumstance, such as division by zero, running out of memory, not being able to open a file, trying to initialize an object to an impossible value, using an out-of-bounds index to a array, and etc.

Contd..

- **What is exception handling?**
  - Detecting unusual or unexpected conditions of the program at run time and taking preventive measure is called exception handling.
  - The purpose of exception handling mechanism is to provide a means to detect and report an exceptional circumstance to that appropriate action can be taken.
  - In c++, exception handling attempts to increase reliability by designing system to continue to provide service in spite of the presence of faults.

Exception handling mechanism

- Exception handling mechanism in C++ is basically built upon three keywords:
  - try,
  - throw and
  - catch

Syntax:

```
try {
  // Block of code to try
  throw exception; // Throw an exception when a problem arises
}
catch () {
  // Block of code to handle errors
}
```

Contd..

- try block:
  - The keyword try is used to surround a block of statements, which may generate exceptions or a function called from that code generates exception. This block of statements is known as try block.
  - The try block is responsible for testing the existence of exceptions.
  - When an exception exist then the normal program flow is interrupted and the program control is transferred to the appropriate catch block that matches the type of the object thrown as exception.

Contd..

- The syntax of try construct is as follows:

```
try
{
    //code that raise exception or

    //call to a function that raise exception

    if (operation fails)
        throw exception;
}
```

Contd..

- throw block:
  - When a problem is detected during the execution of code within try block, an exception is raised using keyword throw.
  - A temporary object of some type is initialized by throw-expression. The type of object is used in matching the catch block.
  - The syntax of the throw construct is as follows:

```
throw [obj];
```

Contd..

- catch block(exception handler):
  - The raised exceptions are handled by the catch block.
  - This exception handler is indicated by the keyword catch.
  - The catch construct must be used immediately after the try block.
  - The syntax of catch construct is as follows:

```
catch(type exception)
{
    //code to be executed in case of exception
}
```

Contd..

- A typical exception handling code have the following pattern:

```
function1()
{
    //.........
    if(operation_fails)
        throw obj1;
}
function2()
{
        try
    {
        //........
        function1();

        if (operation fails)
            throw obj2;
    }
    catch(type1 obj1)
    {
        //handle appropriate actions
    }
    catch(type2 obj2)
    {
        //handle appropriate actions
    }
    //......
}
```

Contd..

Example1:

```
//try block throwing exception
#include<iostream>
using namespace std;
int main()
{
    int a;
    int b;
    cout<<"Enter the value of a and b:"<<endl;
    cin>>a>>b;
    try
    {
        if(b==0)
            throw b;
        else
            cout<<"Result="<<(float)a/b;
    }
    catch(int i)
    {
        cout<<"Divide by zero exception:b="<<i;
    }
    return 0;
}
```

First Execution:

```
Enter the value of a and b:
50
5
Result=10
--------------------------------
```

Second Execution:

```
Enter the value of a and b:
5
0
Divide by zero exception:b=0
--------------------------------
```

Contd..

- **Example2:**

```
//Function invoked by try block thowing exception
#include<iostream>
using namespace std;
void divide(int a, int b)
{
        if(b==0)
            throw b;
        else
            cout<<"Result="<<(float)a/b;
}
int main()
{
    int a;
    int b;
    cout<<"Enter the value of a and b:"<<endl;
    cin>>a>>b;
    try
    {
        divide(a,b);
    }
    catch(int i)
    {
        cout<<"Divide by zero exception:b="<<i;
    }
    return 0;
}
```

First Execution:

```
Enter the value of a and b:
50
5
Result=10
--------------------------------
```

Second Execution:

```
Enter the value of a and b:
5
0
Divide by zero exception:b=0
--------------------------------
```

Multiple exceptions:

- In a program there is a chance of errors occurring more than once at runtime.
- In such a case, C++ permits to design functions to throw as many exceptions as needed.

Example:

```
// program with multiple exceptions and multiple handlers
#include<iostream>
using namespace std;
int main()
{
    int size;
    cout<<"Enter size:"<<endl;
    cin>>size;
    try
    {
        char *mystring;
        if (size<=0)
            throw 's';
        mystring = new char [size];
        for(int n=0;n<=100;n++)
        {
            cout<<n<<endl;
            if(n>size-1)
                throw n;
        }
    }
    catch(int i)
    {
        cout<<"Exception!";
        cout<<"Index"<<i<<"is out of range"<<endl;
    }
    catch(char c)
    {
        cout<<"Exception!";
        cout<<"size must be non-zero positive number";
    }
    return 0;
}
```

```
Enter size:
-10
Exception!size must be non-zero positive number
--------------------------------
```

```
Enter size:
20
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Exception!Index20is out of range
```

```
#include <iostream>
using namespace std;
int main(){
long int a,b,sqr,c[5];
cout<<"enter a positive number"<<endl;
cin>> a;

try{
    if(a<0){
        throw(a);}

    else{
    sqr = a*a;
    cout<<"The Square is "<<sqr<<endl;
    cout<<"Enter position to store square: "<<endl;
    cin>> b;
    if (b<0 ||  b>5){
        throw 's';
    }
    else{
        c[b] = sqr;
        cout<<"Stored successfully at "<< b << "Value ="<<c[b]
            <<endl;
    }

    }
}
catch(long int a){
cout<<"\n You entered: "<<a<<" Please enter a positive number"
    <<endl;}
catch(char b){
    cout<<" Index must be in range 0 to 4"<<endl;
}
    return 0;
}
```

Exception with arguments:

- throw objects(exception) with information about the cause of exception.
- The exception with argument helps the programmer to know what bad value actually caused the exception.

Contd..

- To throw exception with arguments define the exception class with members. i.e., Declare exception classes to throw object with information of the cause of the error.
- The catch block mention type and object to catch and get the object thrown from the exception as:

```
try
{
    //throw object
}
catch(class_name obj)
{
    // handle the exception as according to the value of obj
    //with the help of the exception object in parameter, the
    //description information of the cause of the exception
    //can be provided to the exception handler
}
```

Example:

```
//Exception with arguments
#include<iostream>
#include<cstring>
#include<cmath>
using namespace std;
class number
{
    private:
        double num;
    public:
        class Neg
        {
            public:
                double value;
                char description[20];
                Neg()
                {
                    value=0;
                    description[0]='\0';
                }
                Neg(double v, char *desc)
                {
                    value=v;
                    strcpy(description, desc);
                }
        };
        void readnum()
        {
            cout<<"Enter number:";
            cin>>num;
        }
        double sqroot()
        {
            if(num<0)
                throw Neg(num,"Negative Number");
            else
                return (sqrt(num));
        }
};
int main()
{
    number n1;
    double r;
    n1.readnum();
    try
    {
        r=n1.sqroot();
        cout<<"square root:"<<r<<endl;
        cout<<"success!";
    }
    catch(number::Neg n)
    {
        cout<<"Exception!"<<endl;
        cout<<"Trying to find square root of "<<n.value<<endl;
        cout<<"Error description:"<<n.description;
    }
    return 0;
}
```

```
Enter number:25
square root:5
success!
```

```
Enter number:-25
Exception!
Trying to find square root of -25
Error description:Negative Number
--------------------------------
```

Why Exception Handling?

- Following are main advantages of exception handling over traditional error handling.
  - Separation of Error Handling code from Normal Code
  - Methods can handle any exceptions they choose
  - Grouping of Error Types

Contd..

- Separation of Error Handling code from Normal Code:
  - In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

Contd..

- Methods can handle any exceptions they choose:
  - A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.

    In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

Contd..

- **Grouping of Error Types:**
  - In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Homework

- What is template? State and explain the importance of template.
- What is template? Explain the function template with example.
- What is function template? Differentiate it with class template.
- Write a program to find sum of two number with different data types by using template functions.
- Explain the exception handling with example.
- Discuss different keywords used in exception handling.
- Why do we need exceptions? Explain “exceptions with arguments” with suitable program.
- What are the advantages of exception handling over conventional error handling.

End of Unit 7