

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
CL 103 - COMPUTER PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms. Mahrukh Khan, Ms. Ammara Yaseen, Ms. Tooba Ali, Ms. Maham Mobin, Mr. Muhammad Irfan Ayub

Email: basit.jasani@nu.edu.pk , mahrukh.khan@nu.edu.pk, ammara.yaseen@nu.edu.pk, tooba.ali@nu.edu.pk, maham.mobin@nu.edu.pk, muhammad.irfan@nu.edu.pk

Lab # 11

Outline

- Generic Programming
 - Templates
 - Function Templates
 - Class Templates
 - Examples
 - Exception Handling (Basics)
 - Exercise
-

GENERIC PROGRAMMING

A generic program abstraction (function, class) defines a general set of operations that will be applied to various types of data.

TEMPLATES

In C++ generic programming is done using templates. The normal meaning of the word "template" accurately reflects its use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed.

Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data upon which the function or class operates is specified as a parameter. Thus, one function or class can be used with several different types of data without having to explicitly recode specific versions for each data type.

KINDS OF TEMPLATES

- Function Templates
- Class Templates

FUNCTION TEMPLATES

Through a generic function, as single general procedure can be applied to a wide range of data. For example, the insertion sort sorting algorithm is the same whether it is applied to an array of integers or an array of floats. It is just that the type of the data being sorted is different. By creating a generic function, you can define the nature of the algorithm, independent of any data. Once you have done this, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function. In essence, when you create a generic function you are creating a function that can automatically overload itself.

DECLARATION

```
Template < class T>
Void funName (T x);
// OR
Template <typename T >
Void funName (T x);
// OR
Template < class T, class U ...>
Void funName (T x, U y ...);
```

All function-template definitions begin with the keyword template followed by template parameters enclosed in angle brackets (< and >); each template parameter that represents a type must be preceded by keyword class or type name. Keywords type name and class used to specify function-template parameters mean "any fundamental type or user-defined type."

EXAMPLE (FUNCTION TEMPLATE)

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

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

int main()
{
    int iResult = largest(2, 4);
    cout<<iResult<<endl;

    float fResult = largest(3.5, 3.9);
    cout<<fResult<<endl;

    char cResult = largest('A', 'Z');
    cout<<cResult<<endl;

    return 0;
}
```

```
4
3.9
Z
```

```
-----
Process exited after 0.01465 seconds with return value 0
Press any key to continue . . .
```

OUTPUT: EXAMPLE (FUNCTION TEMPLATE)

A TEMPLATE FUNCTION WITH TWO GENERIC TYPES

More than one generic data type in the **template** statement can be defined by using a comma-separated list. For example, this program creates a template function that has two generic types.

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

template<class type1, class type2>

void myfunc(type1 x, type2 y)
{
    cout<< x << ' ' << y << '\n';
}

int main()
{
    myfunc(10, "I like C++");
    myfunc(98.6, 19);
    return 0;
}
```

The placeholder types “type1 and type2” are replaced by the compiler with the data types int and char *, and double and long, respectively, when the compiler generates the specific instances of myfunc() within main().

```
10 I like C++
98.6 19

-----
Process exited after 0.06432 seconds with return value 0
Press any key to continue . . . _
```

OUTPUT: TEMPLATE FUNCTION WITH TWO GENERIC TYPES

EXPLICIT TYPE PARAMETERIZATION

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

template <typename T, typename U>
T GetInput( U u )
{
    cout<<u<<endl;
    return u;
}

int main()
{
    double d = 10.5674;
    double i = GetInput<int>( d );
    cout<<i;
    return 0;
}
```

USER DEFINED SPECIALIZATION

- A template may not handle all the type successfully.
- Explicit specializations need to be provided for specific type(s)

EXAMPLE (USER DEFINED SPECIALIZATION)

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

template<typename T >
bool isEqual( T x, T y )
{
return ( x == y );
}

template<const char* >
bool isEqual(const char* x, const char* y )
{ return ( strcmp(x,y)==0);}

int main()
{
cout<<isEqual( 5, 6 )<<endl;
cout<<isEqual( 7.5, 7.5 )<<endl;
cout<<isEqual( "abc", "xyz" )<<endl;
return 0;
}
```

CLASS TEMPLATES

- Just as we can define function templates, we can also define class templates.
- A single class template provides functionality to operate on different types of data
- Facilitates reuse of classes.
- A class or class template can have member functions that are themselves templates
- For example, A **Vector** class template can store data elements of different types

DECLARATION

The general form of a generic class declaration is shown here:

```
template<class type>
class class-name
{
}
```

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. More than one generic data type can be defined by using a comma-separated list.

EXAMPLE (CLASS TEMPLATE)

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

template<class T>
class myClass
{
public:
    T element;

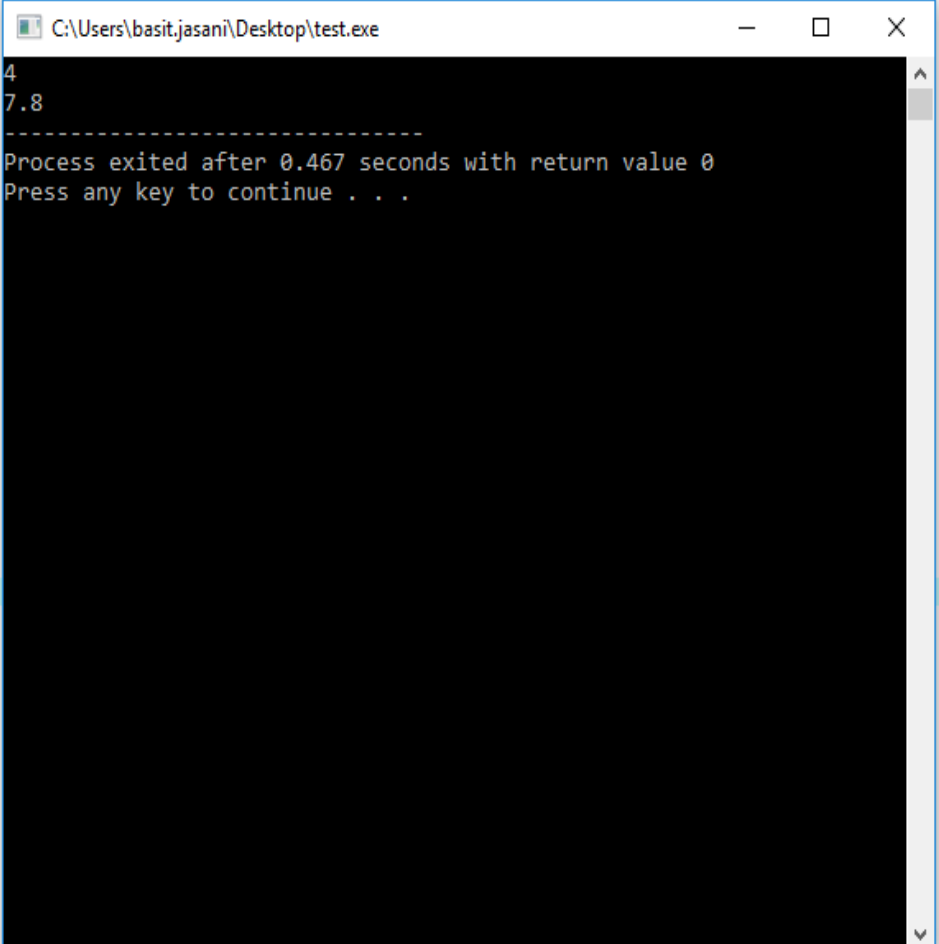
    //constructor
    myClass(T element)
    {
        this->element = element;
    }

    //increment function
    T increment()
    {
        return ++element;
    }
};

int main()
{
    myClass<int> iOb(3);
    int iResult = iOb.increment();
    cout << iResult << endl;

    myClass<float> fOb(6.8);
    float fResult = fOb.increment();
    cout<<fResult;

    return 0;
}
```



```
C:\Users\basit.jasani\Desktop\test.exe
4
7.8
-----
Process exited after 0.467 seconds with return value 0
Press any key to continue . . .
```

CLASS TEMPLATE SPECIALIZATION

- Like function templates, a class template may not handle all the types successfully
- Explicit specializations are provided to handle such types.

EXAMPLE (CLASS TEMPLATE SPECIALIZATION)

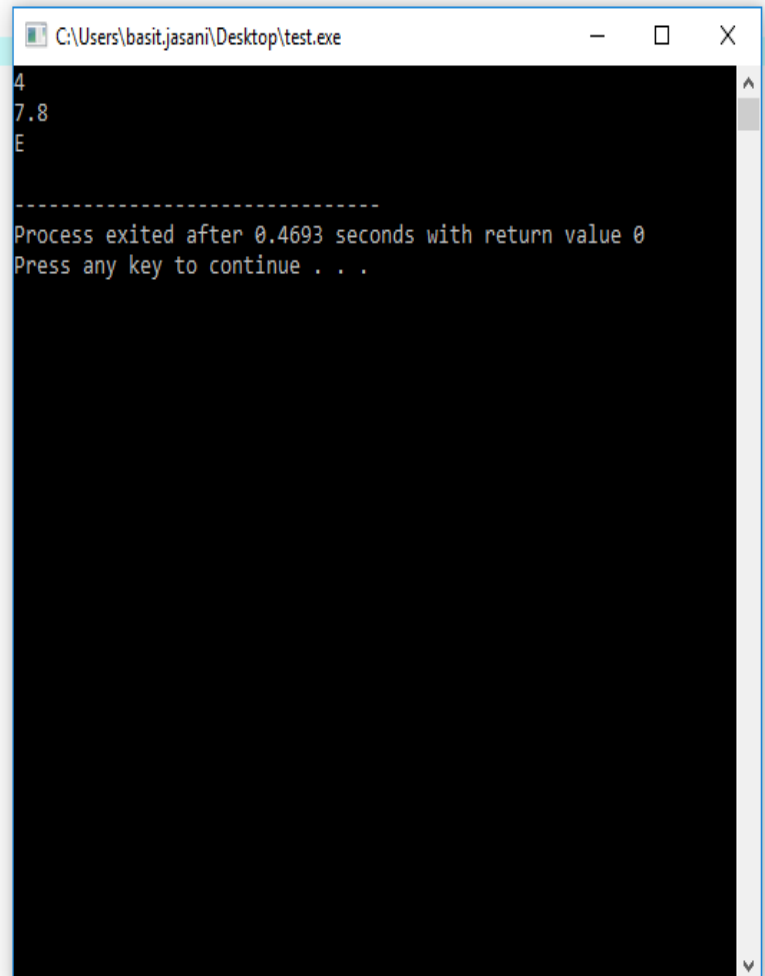
```
template<class T>
class myClass
{
public:
    T element;
    //constructor
    myClass(T element)
    {
        this->element = element;
    }
    //increment function
    T increment()
    {
        return ++element;
    }
};

//specialized class template
template<>
class myClass<char>
{
public:
    char element;

    myClass(char element)
    {
        this->element = element;
    }
    char increment()
    {
        if(element >= 'a' && element <= 'z')
            element += 'A'-'a'; //converting from lowercase to uppercase

        return element;
    }
};

int main()
{
    myClass<int> iOb(3);
    int iResult = iOb.increment();
    cout << iResult << endl;
    myClass<float> fOb(6.8);
    float fResult = fOb.increment();
    cout << fResult << endl;
    myClass<char> cOb('e');
    char cResult = cOb.increment();
    cout << cResult << endl;
    return 0;
}
```



Exception Handling

One of the advantages of C++ over C is Exception Handling. C++ provides following specialized keywords for this purpose.

- **Try:** represents a block of code that can throw an exception.
- **Catch:** represents a block of code that is executed when a particular exception is thrown.
- **Throw:** Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

Why Exception Handling?

Following are main advantages of exception handling over traditional error handling.

1) 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.

2) Functions/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)

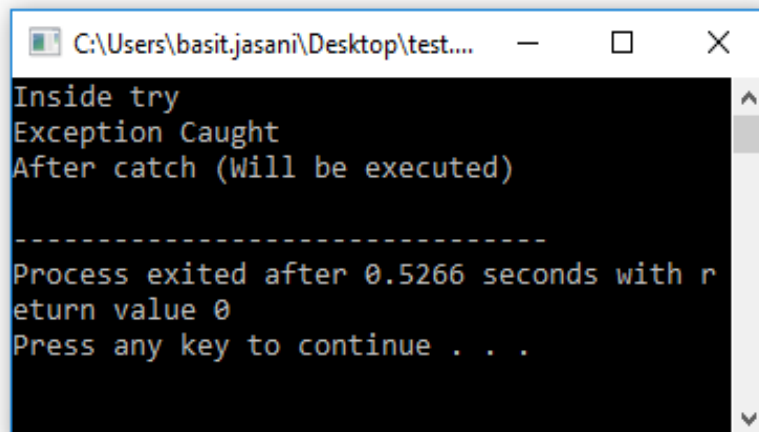
Simple Example:

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

int main()
{
    int x = -1;

    // Some code
    cout << "Before try \n";
    try {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw (Never executed) \n";
        }
    }
    catch (int x ) {
        cout << "Exception Caught \n";
    }

    cout << "After catch (Will be executed) \n";
    return 0;
}
```



There is a special catch block called 'catch all' catch(...) that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so catch(...) block will be executed.


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

int main()
{
    try {
        throw 10;
    }
    catch (char *excp) {
        cout << "Caught " << excp;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}
```

```
C:\Users\basit.jasani\Desktop\test.exe
Default Exception
-----
Process exited after 0.4529 seconds with return value 0
Press any key to continue . . .
```

If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch a char.

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

int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught ";
    }
    return 0;
}
```

```
C:\Users\basit.jasani\Desktop\test.exe
terminate called after throwing an instance of 'char'

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
-----
Process exited after 4.579 seconds with return value 3
Press any key to continue . . .
```

In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using “throw;”

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

int main()
{
    try {
        try {
            throw 20;
        }
        catch (int n) {
            cout << "Handle Partially ";
            throw; //Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

```
Select C:\Users\basit.jasani\Desktop\test.exe
Handle Partially Handle remaining
-----
Process exited after 0.4557 seconds with return value 0
Press any key to continue . . .
```

LAB 11 EXERCISES

QUESTION#1

Create a class Inventory that contains item code, description, rate per unit and total units in stock.

- a) Create 5 objects of inventory.
- b) Create a menu based program to handle data file to store these five objects.
- c) Also add a function to edit the stock of any given item.
- d) Add a function that displays the details of the item whose code is passed as a parameter.
- e) Add an overloaded version of the function details () that print all the items with their details.

QUESTION#2

Implement bubble sort algorithm using function template and show the results for the following arrays

- 1) 7, 5, 4, 3, 9, 8, 6
- 2) 4.3, 2.5, -0.9, 100.2, 3.0

QUESTION#3

Create a class template with two generic data types to print their addition. Show the results for following types:

- **int** and **double** for example: (10,0.23) would print 10.23
- **char*** and **char***
For example: ("Now", "Then") would print NowThen

QUESTION#4

Create a class containing a function template capable of returning the sum of all the elements in an array being passed as parameter. Show the results for two arrays, one integer type and the other double type.