TEMPLATE

Course Code: CSC1102 &1103 Course Title: Introduction to Programming

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**14**	**Week No:**	**11(1X1.5)**	**Semester:**	
Lecturer:	*Name & email*				

Limitation of Traditional Function and Classes

- Need to specify the type of all parameters

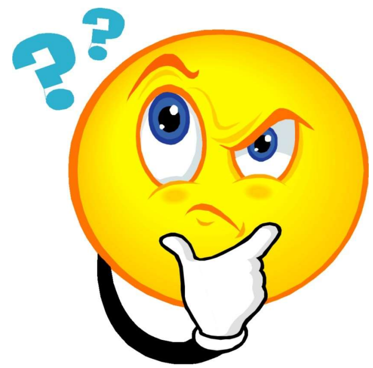
For Example,

We want to find out the addition between two integer type values

```
int maxV(int x, int y)
{
      return (x+y);
}
```

```
float maxV(float x, float y)
{
      return (x+y);
}
```

What would happen ?
If we want to find out the maximum values between two float type values or character type values



Function Overloading ???

Only Changes the type of Parameter

```
char maxV(char x, char y)
{
    return (x > y) ?  x : y;
}
```

```
double maxV(double x, double y)
{
    return (x > y) ?  x : y;
}
```

Limitations of Function Overloading

Changing only the type of Parameters

- can become a maintenance headache
- Time waster
- Violates the general programming guidelines
- Increases code duplication

Welcome to the world of TEMPLATES

What is Template Function

Dictionary Meaning:

A Template is a model that serves as a pattern for creating similar objects

For Example,

- Cut Out a shape of any letter i.e. **J**
- Place the above stencil on the top of any object
- Spray any color through the hole

Very quickly, you will produce stenciled pattern in many different colors

In C++, **FUNCTION TEMPLATES** are functions that serve as a pattern for creating other similar functions.

Basic Idea is to

- ✓ Create a function without having to specify the exact type (s) of some or all of the variables

we define the function using **placeholder types**

Called Template type Parameters

Lets Create Function Templates

Look at the int version of maxV function again !!!

```
int maxV(int x, int y)
{
    return (x > y) ?  x : y;
}
```

3 places where specific type has been used

We are going to replace them with **placeholder types**

***** As there is only one type of Parameter, We need only one type of Placeholder**

Lets Create Function Templates

Convert this to Template Function

```
T maxV(T x, T y)
{
    return (x > y) ? x : y;
}
```

Can use any Letter or words

It won't compile because compiler does not know what **T** is

Compiler needs to know Two things

- This is a Template Function
- **T** is a Placeholder type

We can do both of those things in one line, called a **template parameter declaration**

```
template<typename  T> // This is template parameter declaration
T maxV(T x, T y)
{
    return (x > y) ?  x : y;
}
```

let’s take a slightly closer look at the template parameter declaration

template

template parameters

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

- place all of parameters inside angled brackets (**<>**)
- Use either the keyword ***typename*** or ***class***

A Complete Program

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

template<typename  T>
T maxV(T x, T y)
{
                return (x > y) ?  x : y;
}

int main()
{
    cout<<maxV<int>(3 , 6)<<endl;
     cout<<maxV<double>(9.5 , 7.4)<<endl;
     cout<<maxV<char>('f' , 'r')<<endl;
}
```

Output

6

9.5

r

Take another example

A templated function of Summation of two numbers

```
template<typename R, typename S>
R sum(R x, S y)
{
    return x + y;
}
```

A Complete Program

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

template<typename  R, typename   S>
R sum(R x, S y)
{
      return x + y;
}

int main()
{
               cout<<sum<int,int>(3 , 6)<<endl;
               cout<<sum<double,int>(4.5 , 7)<<endl;
               cout<<sum<int,double>(6 , 8.4)<<endl;
               cout<<sum<double,double>(3.2 , 6.8)<<endl;
}
```

Output

9

11.5

14

10

How it works...

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

template<typename  T>
T maxV(T x, T y)
{
        return (x > y) ?  x : y;
}

int main()
{
        cout<<maxV<int>(3 , 6)<<endl;
    cout<<maxV<double>(9.5 , 7.4)<<endl;
        cout<<maxV<char>('f' , 'r')<<endl;
}
```

Templated Function

Encounters a call to the Templated Function

Creates Template Instance

```
int maxV(int x, int y)
{
   return(x > y) ? x : y;
}
```

Encounters a call to the Templated Function

Creates Template Instance

```
char maxV(char x, char y)
{
   return(x > y) ? x : y;
}
```

Key points of Templated Functions

- It only needs to create one template instance per set of unique type parameters
- If you create a template function but do not call it, no template instances will be created
- Template functions will work with both built-in types (e.g. char, int, double, etc…) and classes
- Any operators or function calls in your template function must be defined for any types the function template is instantiated for.

Key points of Templated Functions

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

template<typename  R, typename   S>
R sum(R x, S y)
{
   return x + y;
}

R sub(R x, S y)
{
   return x - y;
}
int main()
{
             cout<<sum<int, int>(3,6)<<endl;
     cout<<sub<double, int>(4.5,7)<<endl;
}
```

Templated Function

Not Templated Function

It won't Compile

Lets Solve it…

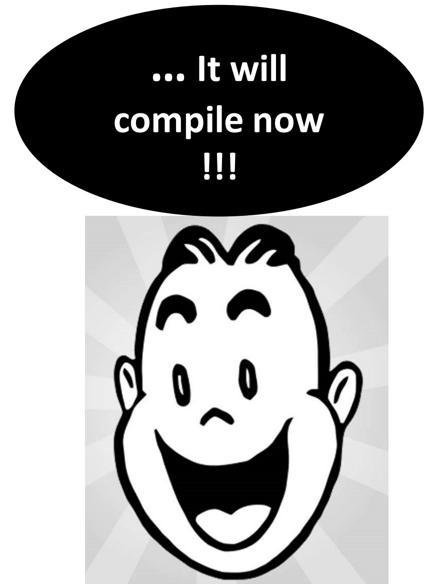
```
#include<iostream>
  using namespace std;

template<typename  R, typename   S>
R sum(R x, S y)
{
   return x + y;
}
```

Templated Function

```
template<typename  R, typename   S>
R sub(R x, S y)
{
   return x - y;
}
int main()
{
            cout<<sum<int, int>(3,6)<<endl;
     cout<<sub<double, int>(4.5,7)<<endl;
}
```

Templated Function



Another Example

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

template<typename P, typename R>
class Triangle
{
   P height;
   R length;
public:
   Triangle(P ht, R len)
   {
      height=ht;
      length=len;
   }
   P area()
   {
      return 0.5*height*length;
   }};
```

```
int main()
{
   Triangle <double, double>t1(3,5);
   Triangle <double, int>t2(4.8,7.6);
   Triangle <int, int>t3(4,9);
   cout<<t1.area()<<endl;
   cout<<t2.area()<<endl;
   cout<<t3.area()<<endl;
}
```

Output

7.5

16.8

18

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

template<typename P, typename R>
class Triangle
{
  double height;
  double length;
public:
   Triangle(double ht, double len)
   {
      height=ht;
      length=len;
   }
   double area()
   {
      return 0.5*height*length;
   }
};
```

How it works...

```cpp
int main()
{
   Triangle <double, double>t1(3,5);
   Triangle <double, int>t2(4.8,7.6);
   Triangle <int, int>t3(4,9);
   cout<<t1.area()<<endl;
   cout<<t2.area()<<endl;
   cout<<t3.area()<<endl;
}
```

How it works…

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

template<typename P, typename R>
class Triangle
{
  double height;
  int length;
public:
  Triangle(double ht, int len)
  {
    height=ht;
    length=len;
  }
  double area()
  {
    return 0.5*height*length;
  }
};
```

```
int main()
{
  Triangle <double, double>t1(3,5);
  Triangle <double, int>t2(4.8,7.6);
  Triangle <int, int>t3(4,9);
  cout<<t1.area()<<endl;
  cout<<t2.area()<<endl;
  cout<<t3.area()<<endl;
}
```

Disadvantages of Template Functions

- Historically, some compilers exhibited poor support for templates. So, the use of templates could decrease code portability.
- Many compilers lack clear instructions when they detect a template definition error.
- Since the compiler generates additional code for each template type, indiscriminate use of templates can lead to code bloat, resulting in larger executables.
- It can be difficult to debug code that is developed using templates. Since the compiler replaces the templates, it becomes difficult for the debugger to locate the code at runtime.

THANK YOU