

# **Lecture 16**

## **OOP**

**Khola Naseem**  
**khola.naseem@uet.edu.pk**

# Function Templates:

- Overloaded functions, The only difference were the types that appear in the parameter list.
- It seems an unnecessary overhead to have to write the same code over and over, just because it has to work for different types.
- And indeed it is. In such situations you can write the code just once, as a **function template**.
- A function template itself it is not a definition of a function; it is a blueprint or a recipe for defining an entire family of functions. A function template is a parametric function definition, where a particular function instance is created by one or more parameter values.

# Function Templates:

## ➤ Template:

The `template` keyword identifies this code as a template.

The `typename` keyword identifies `T` as a type. You put the template parameters between angled brackets after the template keyword. They are separated by commas if there is more than one.

This `T` is a parameter for the template. It identifies where the type for a particular instance has to be substituted in the code. In this case the return type and both parameter types are to be replaced.

```
template <typename T> T larger(T a, T b)
{
    return a > b ? a : b;
}
```

Every occurrence of `T` is replaced by an actual type when an instance of the template is created. Wherever `T` appears in the template definition, it will be replaced by a specific type.

# Function Templates:

- The compiler uses a function template to generate a function definition when necessary.
- If it is never necessary, no code results from the template. A function definition that is generated from a template is an instance or an instantiation of the template.
- The parameters of a function template are usually data types, where an instance can be generated for a parameter value of type `int`, for example, and another with a parameter value of type `string`.
- Templates are powerful features of C++ which allows us to write generic programs.

# Function Templates:

- This is followed by a pair of angle brackets that contains a list of one or more template parameters.
- In this case, there's only one, the parameter T.
- T is commonly used as a name for a parameter because most parameters are types, but you can use whatever name you like for a parameter; names such as type, MY\_TYPE, or Comparable are equally valid.
- The typename keyword identifies that T is a type. T is hence called a template type parameter.
- You can also use the keyword class here the keywords class and typename are synonymous in this context but we prefer typename.
- The rest of the definition is similar to a normal function except that the parameter name T is sprinkled around. The compiler creates an instance of the template by replacing T throughout the definition with a specific type.
- The type assigned to a type parameter T during instantiation is called a template type argument. You can position the template in a source file in the same way as a normal function definition; you can also specify a prototype for a function template.

# Function Templates:

- In this case, it would be as follows:

```
template<typename T> T larger(T a, T b); // Prototype for function template
```

# Creating Instances of a Function Template

- In this case, it would be as follows:

```
template<typename T> T larger(T a, T b); // Prototype for function template
```

- The compiler creates instances of the template from any statement that uses the larger() function. Here's an example:

```
std::cout << "Larger of 1.5 and 2.5 is " << larger(1.5, 2.5) << std::endl;
```

# Creating Instances of a Function Template

➤ Here's an example:

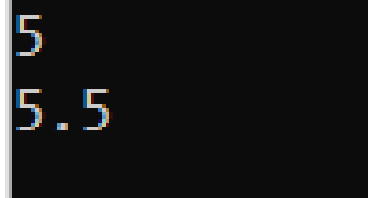
```
template <typename T>
T add(T num1, T num2) {
    return (num1 + num2);
}

int main() {

    int result1;
    double result2;
    // calling with int parameters
    result1 = add(2, 3);
    cout << result1 << endl;

    // calling with double parameters
    result2 = add(2.2, 3.3);
    cout << result2 << endl;

    return 0;
}
```



5  
5.5



# Creating Instances of a Function Template

## ➤ Same type:

```
#include <iostream>
using namespace std;
template<typename X> void fun(X a,X b)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}

int main()
{
    int a=4;
    fun(a,12.3);
    void fun (X a, X b)
}
```

## ➤ Error

[Error] no matching function for call to 'fun(int&, double)'

[Note] candidate is:

[Note] template<class X> void fun(X, X)

# Creating Instances of a Function Template

➤ sol:

```
#include <iostream>
using namespace std;
template<typename X> void fun(X a,X b)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}

int main()
{
    int a=4;
    fun<double>(a,12.3);
    return 0;
}
```

```
Value of a is : 4
Value of b is : 12.3

-----
Process exited after 0.1283 seconds with return value 0
Press any key to continue . . .
```

# Template Type Parameters

- The name of a template type parameter can be used anywhere in the template's function signature, return type, and body
- such as T&, const T&, T\*, and T[][3].
  - `template const T& larger(const T& a, const T& b){}`
- Pass default values

```
template <typename ReturnType=double, typename T1, typename T2>  
ReturnType larger(const T1&, const T2&);
```

# Function Templates with Multiple Parameters

- one generic type in the template function by using the comma to separate the list

```
template<typename T1, class T2,.....>
```

```
return_type function_name (arguments of type T1, T2....)
```

```
{
```

```
    // body of function.
```

```
}
```

# Function Templates with Multiple Parameters

- one generic type in the template function by using the comma to separate the list

```
template<typename X, class Y> void fun(X a, Y b)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}

int main()
{
    fun(15,12.3);

    return 0;
}
```

- Output:

```
Value of a is : 15
Value of b is : 12.3
```

# Restrictions of Generic Functions

- Generic functions perform the same operation for all the versions of a function except the data type differs. Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities

```
#include <iostream>
using namespace std;
void fun(double a)
{
    cout<<"value of a is : "<<a<<"\n";
}

void fun(int b)
{
    if(b%2==0)
    {
        cout<<"Number is even";
    }
    else
    {
        cout<<"Number is odd";
    }
}

int main()
{
    fun(4.6);
    fun(6);
    return 0;
}
```

# non-type parameters to templates

- We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of a template. The important thing to note about non-type parameters is, that they must be const.
- The compiler must know the value of non-type parameters at compile time. Because the compiler needs to create functions/classes for a specified non-type value at compile time.
- In the following code , if we replace 10000 or 25 with a variable, we get a compiler error.

# non-type parameters to templates

## ➤ Non-type:

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

template <class T, int max> int arrMin(T arr[], int n)
{
    int m = max;
    for (int i = 0; i < n; i++)
        if (arr[i] < m)
            m = arr[i];

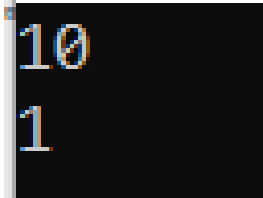
    return m;
}

int main()
{
    int arr1[] = { 10, 20, 15, 12 };
    int n1 = sizeof(arr1) / sizeof(arr1[0]);

    char arr2[] = { 1, 2, 3 };
    int n2 = sizeof(arr2) / sizeof(arr2[0]);

    // Second template parameter
    // constant
    cout << arrMin<int, 10000>(arr1, n1) << endl;
    cout << arrMin<char, 256>(arr2, n2);

    return 0;
}
```





# Generic Functions

- `larger(T a, T b)`
- Calling:
- `larger(3.5,4.5)`
- `larger(7,8)`
- `larger("a","z")` //a and z are strings

# Reference material

## ➤ **For Practice Questions, refer to these books**

- C++ Programming From Problem Analysis To Program Design, 5th Edition, D.S.Malik. Chapter 12.
- C++ How to Program, Deitel & Deitel, 5th Edition, Prentice Hall.
- Object Oriented Programming in C++ by Robert Lafore.
- Object Oriented Software Construction, Bertrand Meyer's
- Object-Oriented Analysis and Design with applications, Grady Booch et al, 3Rd Edition, Pearson, 2007
- Web