SCS1310: Object-Oriented Modelling and Programming

Templates

Viraj Welgama

Polymorphism...

- when multiple functions do similar operations:
  - Function overloading

- when multiple functions do identical operations:
  - Templates

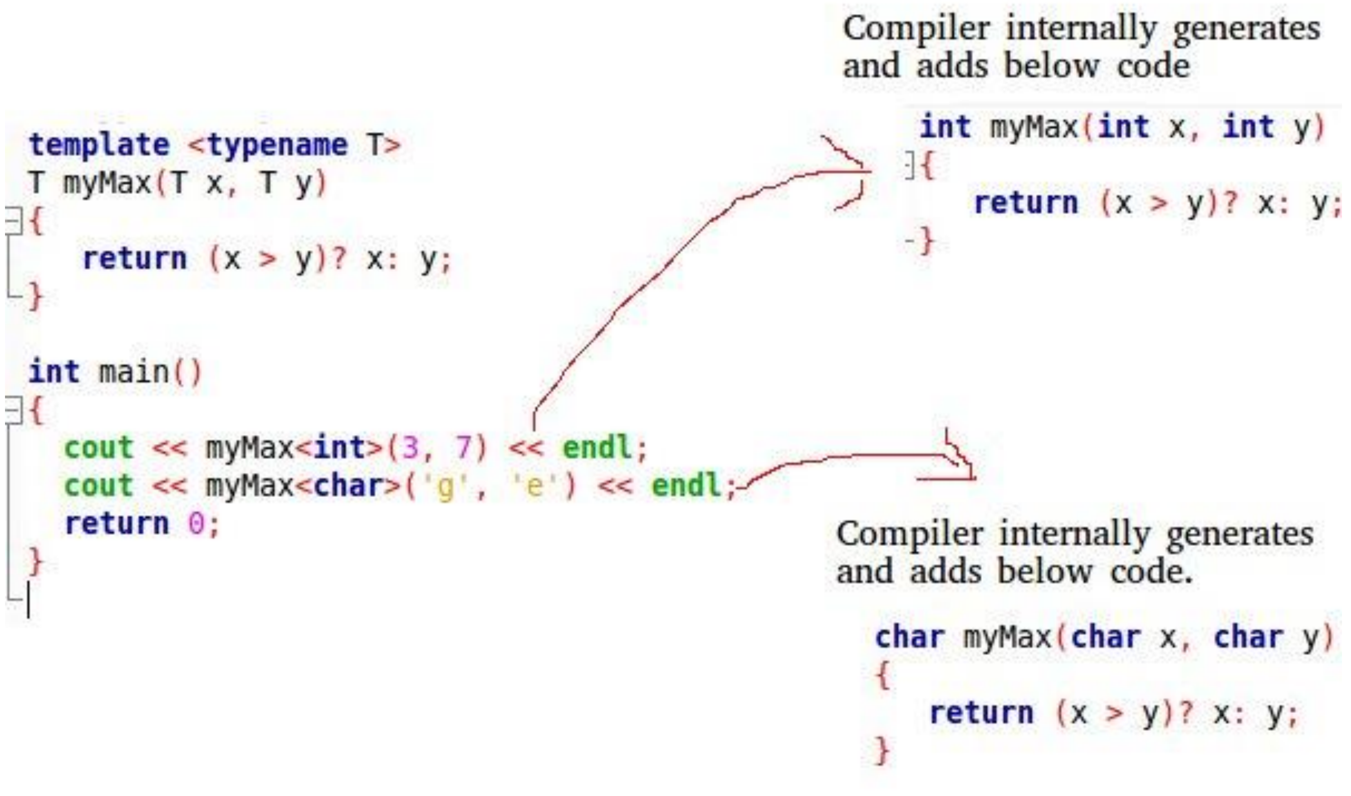
What is a Template?

- A template is a simple and yet very powerful tool in C++.
- The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types.
  - For example, a software company may need sort() for different data types.
  - Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

How templates work?

- Templates are expanded at compiler time.
- This is like macros. The difference is, compiler does type checking before template expansion.
- The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.

How templates work?



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

int main()
{
  cout << myMax<int>(3, 7) << endl;
  cout << myMax<char>('g', 'e') << endl;
  return 0;
}
```

Compiler internally generates and adds below code

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

Compiler internally generates and adds below code.

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

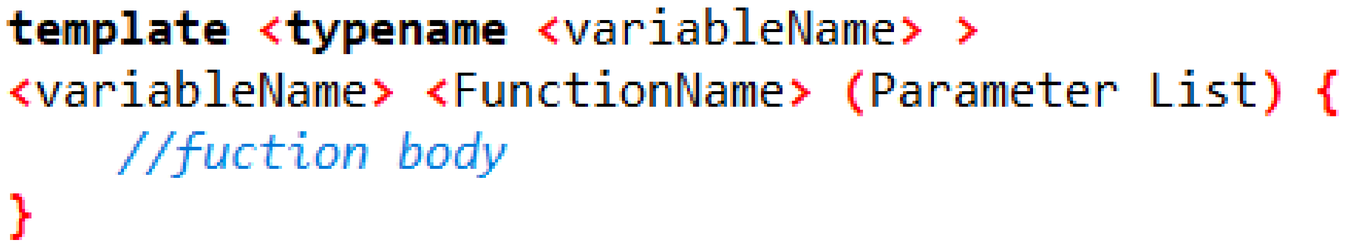
Two Types:

1. Function Templates
2. Class Templates

Function Templates

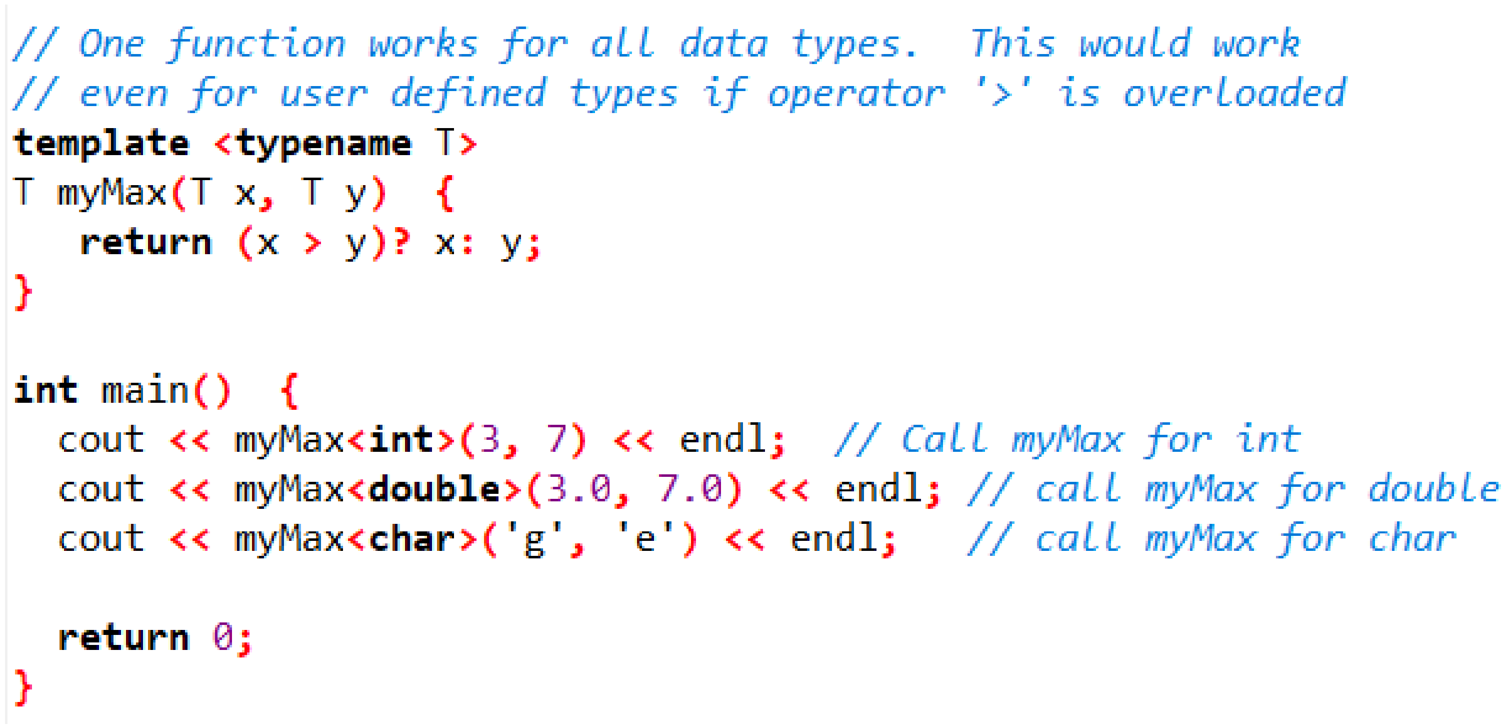
- We write a generic function that can be used for different data types.
  - Examples of function templates are sort(), max(), min(), printArray().

- Syntax:



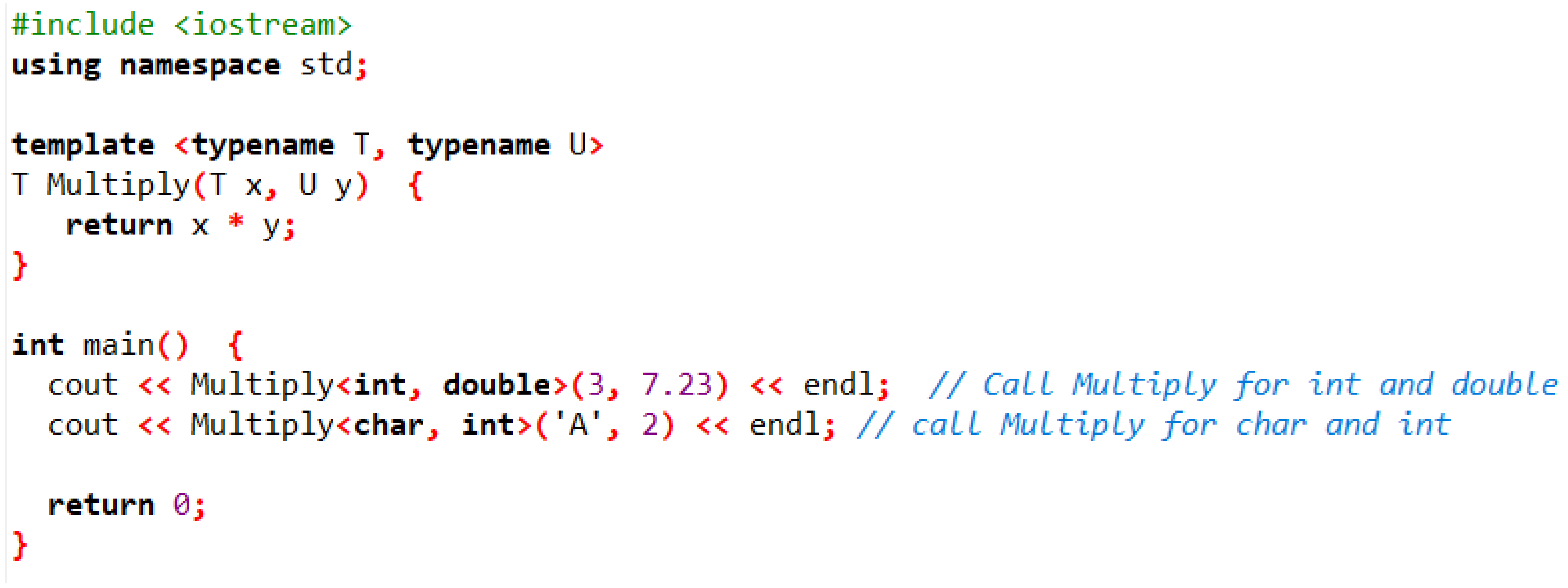
```
template <typename <variableName> >
<variableName> <FunctionName> (Parameter List) {
    //fuction body
}
```

Function Templates: Example



```
// One function works for all data types.  This would work
// even for user defined types if operator '>' is overloaded
template <typename T>
T myMax(T x, T y)  {
   return (x > y)? x: y;
}

int main()  {
  cout << myMax<int>(3, 7) << endl;  // Call myMax for int
  cout << myMax<double>(3.0, 7.0) << endl; // call myMax for double
  cout << myMax<char>('g', 'e') << endl;   // call myMax for char

  return 0;
}
```

Multiple Arguments



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

template <typename T, typename U>
T Multiply(T x, U y)  {
   return x * y;
}

int main()  {
  cout << Multiply<int, double>(3, 7.23) << endl;  // Call Multiply for int and double
  cout << Multiply<char, int>('A', 2) << endl; // call Multiply for char and int

  return 0;
}
```

Templates with non-type parameters

- 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, they must be const.
- The compiler must know the value of non-type parameters at compile time.
  - Because compiler needs to create functions/classes for a specified non-type value at compile time.

Example:

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

template <typename 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, 8, 9, 2};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);

    char arr2[] = {12, 2, 3, 10, 4};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);

    // Second template parameter to arrMin must be a constant
    cout << arrMin<int, 1>(arr1, n1) << endl;
    cout << arrMin<char, 256>(arr2, n2);
    return 0;
}
```

Class Templates

- Like function templates, class templates are useful when a class defines something that is independent of the data type.
  - Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

Template of Array Class

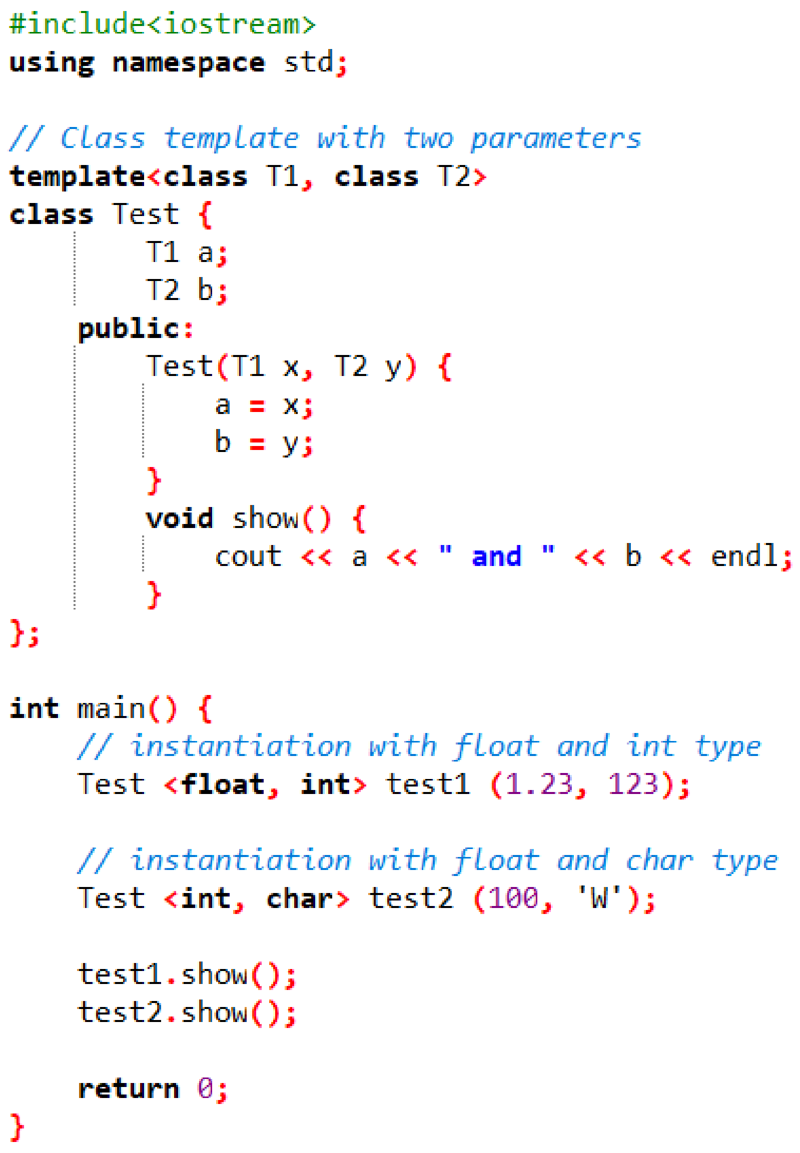
```
#include <iostream>
using namespace std;

template <typename T>
class Array {
private:
    T *ptr;
    int size;
public:
    Array(T arr[], int s);
    void print();
};

template <typename T>
Array<T>::Array(T arr[], int s) {
    ptr = new T[s];
    size = s;
    for(int i = 0; i < size; i++)
        ptr[i] = arr[i];
}

template <typename T>
void Array<T>::print() {
    for (int i = 0; i < size; i++)
        cout<<" "<<*(ptr + i);
    cout<<endl;
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    string starr[4] = {"AB", "XY", "CD", "MN"};
    Array<int> a(arr, 5);
    a.print();
    Array<string> s(starr, 4);
    s.print();
    return 0;
}
```

Class template with multiple parameters

- While creating templates, it is possible to specify more than one type.
- We can use more than one generic data type in a class template
- **Syntax:**

```
template<class T1, class T2, ...>
class classname
{
        ...
        ...
};
```

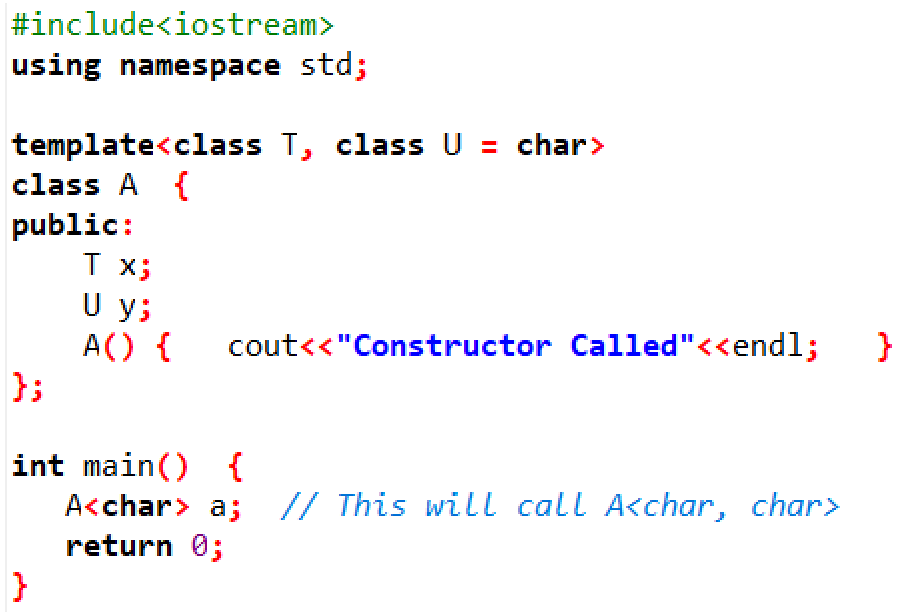
Example:



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

// Class template with two parameters
template<class T1, class T2>
class Test {
        T1 a;
        T2 b;
    public:
        Test(T1 x, T2 y) {
            a = x;
            b = y;
        }
        void show() {
            cout << a << " and " << b << endl;
        }
};

int main() {
    // instantiation with float and int type
    Test <float, int> test1 (1.23, 123);

    // instantiation with float and char type
    Test <int, char> test2 (100, 'W');

    test1.show();
    test2.show();

    return 0;
}
```

Default Values for Template Arguments

- Like normal parameters, we can specify default arguments to templates.
- All default values must be on the rightmost side as in normal functions.



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

template<class T, class U = char>
class A  {
public:
    T x;
    U y;
    A() {   cout<<"Constructor Called"<<endl;   }
};

int main()  {
   A<char> a;  // This will call A<char, char>
   return 0;
}
```