Templates in c++

Templates

Use to create a single function or a class to work with different data types
The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

Function Templates

A function template works in a similar to a normal function, with one key difference.
A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

How to declare a function template?

A function template starts with the keyword template followed by template parameter/s inside < > which is followed by function declaration.

```
template <class T>
T someFunction(T arg)
{
   ... .. ...
}
```

In the above code, T is a template argument that accepts different data types (int, float), and class is a keyword.

You can also use keyword typename instead of class in the above example.

When, an argument of a data type is passed to someFunction( ), compiler generates a new version of someFunction() for the given data type.

```
void swap(int & a,int & b)
{
  int temp;
  temp=a;
  a=b;
  b=temp;
}
```

Listing 9.1 A function to swap two integers

The preceding 'swap' function swaps the values of two integers. A 'swap' function that swaps two floats will have the following definition:

```
void swap(float & a,float & b)
{
  float temp;
  temp=a;
  a=b;
  b=temp;
}
```

Listing 9.2 A function to swap two float type numbers

Example: template for function swap()

```
Template <class T>
void swap (T & a , T & b)
        {       T temp;
                temp=a;
                a=b;
                b=temp;
        }
```

Note: suppose the swap is passed by two integers, the compiler generate an actual definition for the function by replacing each occurrence of T by the keyword int.

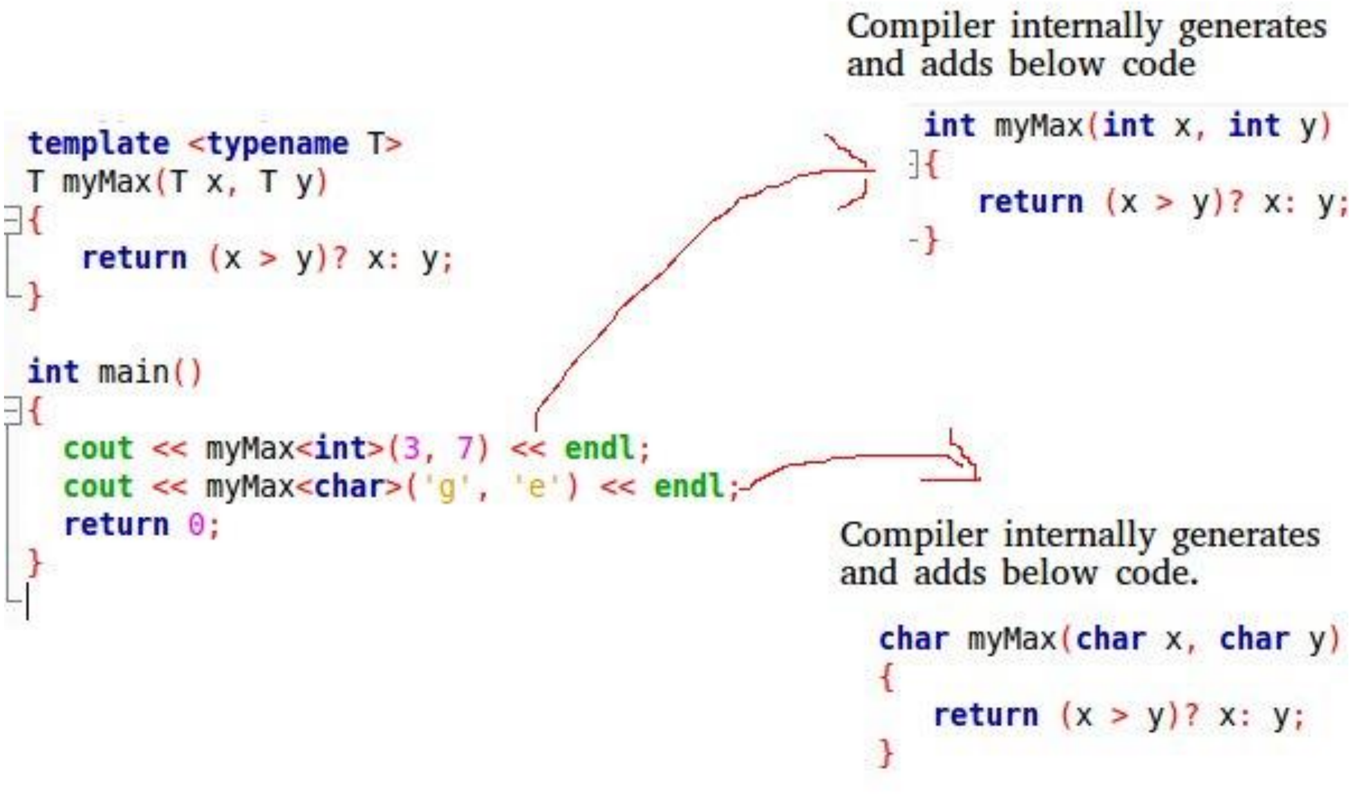
Similarly for float or other data type

How templates work?

Templates are expended at compiler time.

This is like macros but 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.



```
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;
}
```

Program 1:

```
#include <iostream>
#include <string>
using namespace std;
template <typename T>
T Max (T a, T b) {
  return a < b ? b:a;
}
int main () {
  int i = 39;
  int j = 20;
  cout << "Max(i, j): " << Max(i, j) << endl;
  double f1 = 13.5;
  double f2 = 20.7;
  cout << "Max(f1, f2): " << Max(f1, f2) << endl;
  string s1 = "Hello";
  string s2 = "World";
  cout << "Max(s1, s2): " << Max(s1, s2) << endl;
  return 0;
}
```

Program 2:

```
#include <iostream>
using namespace std;
template <typename T>
void Swap(T &n1, T &n2)
{
        T temp;
        temp = n1;
        n1 = n2;
        n2 = temp;
}
int main()
{
        int i1 = 1, i2 = 2;
        float f1 = 1.1, f2 = 2.2;
        char c1 = 'a', c2 = 'b';
        cout << "Before passing data to function template.\n";
        cout << "i1 = " << i1 << "\ni2 = " << i2;
        cout << "\nf1 = " << f1 << "\nf2 = " << f2;
        cout << "\nc1 = " << c1 << "\nc2 = " << c2;
        Swap(i1, i2);
        Swap(f1, f2);
        Swap(c1, c2);
    cout << "\n\nAfter passing data to function template.\n";
        cout << "i1 = " << i1 << "\ni2 = " << i2;
        cout << "\nf1 = " << f1 << "\nf2 = " << f2;
        cout << "\nc1 = " << c1 << "\nc2 = " << c2;
        return 0;
}
```

Class templates

- Need of class templates is similar to the function templates , to handle data of different types.

class template syntax

```
template <class T>
class className
{
   ... .. ...
public:
   T var;
   T someOperation(T arg);
   ... .. ...
};
```

create a class template object

- className<dataType> classObject;

For example:

- – className<int> classObject;
- – className<float> classObject;
- – className<string> classObject;

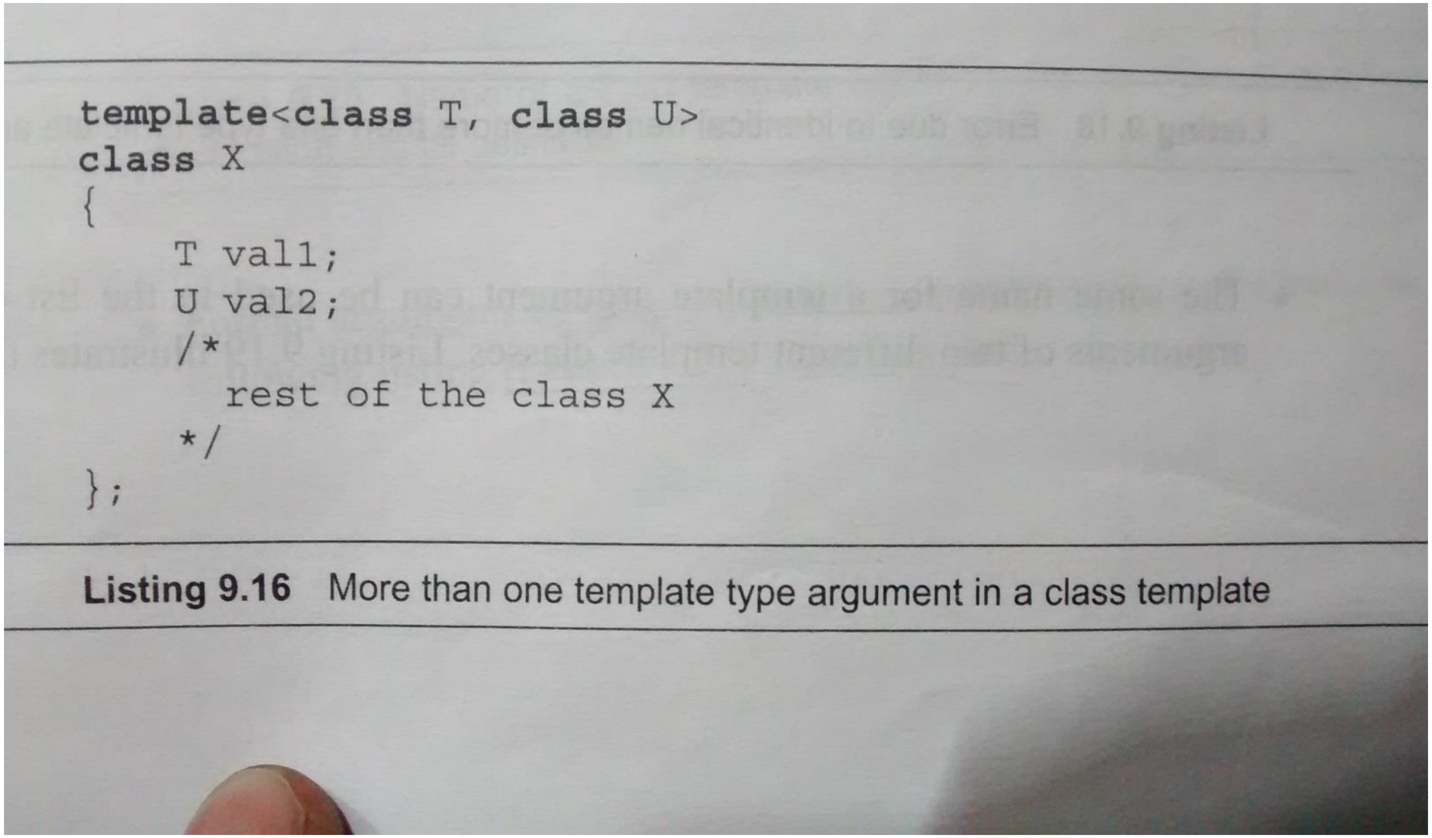
Member function of class templates

- Templates<class T>

```
      void X<T> :: f1(const T&p)
              { //function definition
              }
```

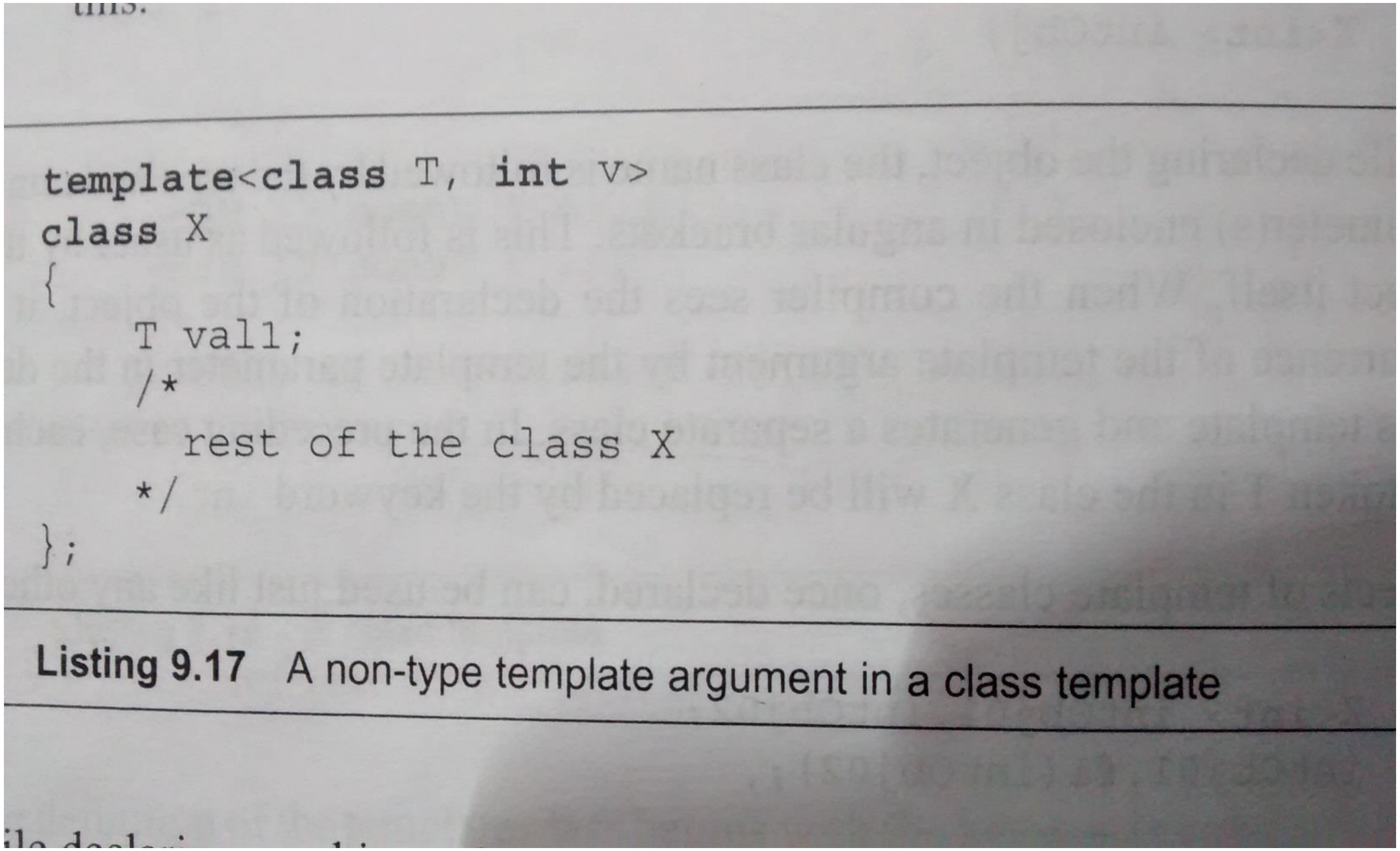
Note: the class name given before the scope resolution operator is followed by the name of all template arguments enclosed in angular bracket.

```
#include <iostream>
using namespace std;
template <class T>
class Calculator
{
private:
        T num1, num2;
public:
        Calculator(T n1, T n2)
        {
                num1 = n1;
                num2 = n2;
        }
        void displayResult()
        {
                cout << "Numbers are: " << num1 << " and " << num2 << "." <<
endl;
                cout << "Addition is: " << add() << endl;
                cout << "Subtraction is: " << subtract() << endl;
                cout << "Product is: " << multiply() << endl;
                cout << "Division is: " << divide() << endl;
        }
        T add() { return num1 + num2; }
        T subtract() { return num1 - num2; }
        T multiply() { return num1 * num2; }
        T divide() { return num1 / num2; }
};
int main()
{
        Calculator<int> intCalc(2, 1);
        Calculator<float> floatCalc(2.4, 1.2);
        cout << "Int results:" << endl;
        intCalc.displayResult();
        cout << endl << "Float results:" << endl;
        floatCalc.displayResult();
        return 0;
}
```



```
template<class T, class U>
class X
{
    T val1;
    U val2;
    /*
      rest of the class X
    */
};
```

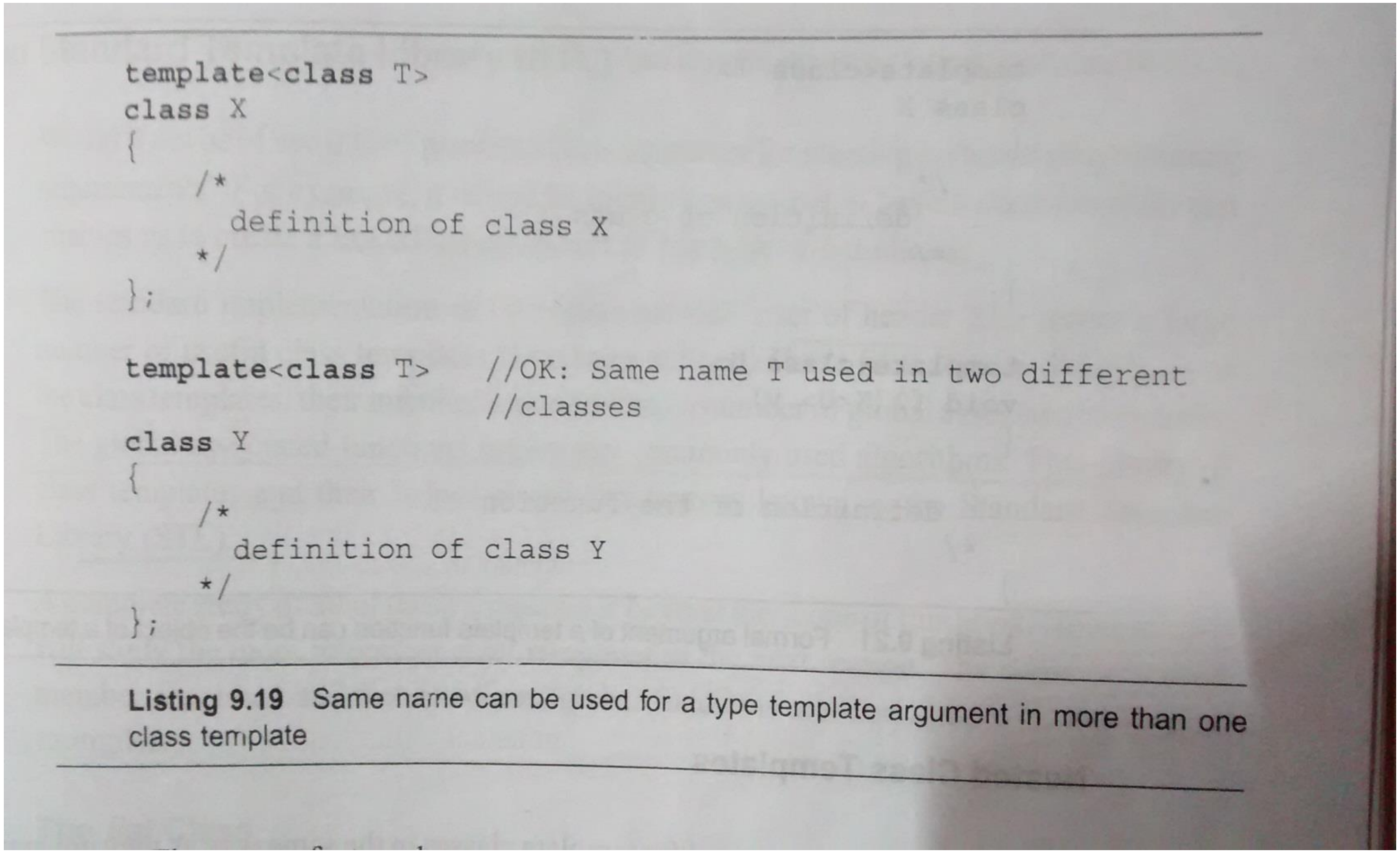
Listing 9.16 More than one template type argument in a class template



this:

```
template<class T, int v>
class X
{
    T val1;
    /*
      rest of the class X
    */
};
```

Listing 9.17 A non-type template argument in a class template



```
template<class T>
class X
{
    /*
      definition of class X
    */
};

template<class T>    //OK: Same name T used in two different
                     //classes
class Y
{
    /*
      definition of class Y
    */
};
```

Listing 9.19 Same name can be used for a type template argument in more than one class template

non-template classes.

```
template<class T>
class A
{
    class B
    {
        T x;   //enclosing template type can be used in the
               //nested class
        /*
          rest of the class B
        */
    };
  /*
    definition of the class A
  */
};
```

Listing 9.22 A nested template class