Chapter 8: Templates

Department of Computer Science and Engineering
Kathmandu University

Instructor: Rajani Chulyadyo, PhD

Contents

Templates

Templates make it possible to use one function or class to handle many different data types.

The template concept can be used in two different ways:

1. with functions and
2. with classes.

Function templates

Suppose we need a function to find the maximum of two numbers.

For integers, we will have the following function.

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

Implementations are exactly same.

For doubles?

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

Function templates

Suppose we need a function to find the maximum of two numbers.

For integers, we will have the following function.

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

For doubles?

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

Rewriting the same function body over and over for different types is time-consuming and wastes space in the listing.

Also, if you find you've made an error in one such function, you'll need to remember to correct it in each function body.

Function templates let you write such a function just once, and have it work for many different data types.

Function templates

Function templates are functions that serve as a pattern for creating other similar functions.

Basic idea: Create a function without having to specify the exact type(s) of some or all of the variables. Instead, we define the function using placeholder types, called **template type parameters**.

Function template syntax

```
template<class T> return_type function_name(parameter list)
{
     //body
}
```

OR

```
template<typename T> return_type function_name(parameter list)
{
     //body
}
```

The `template` keyword signals the compiler that we're about to define a function template. The variable following the keyword `class` or `typename` (`T` in this syntax) is called the *template argument*.

Function template syntax

```
template<class T> return_type function_name(parameter list)
{
     //body
}
```

OR

```
template<typename T> return_type function_name(parameter list)
{
     //body
}
```

The function template itself doesn't cause the compiler to generate any code.

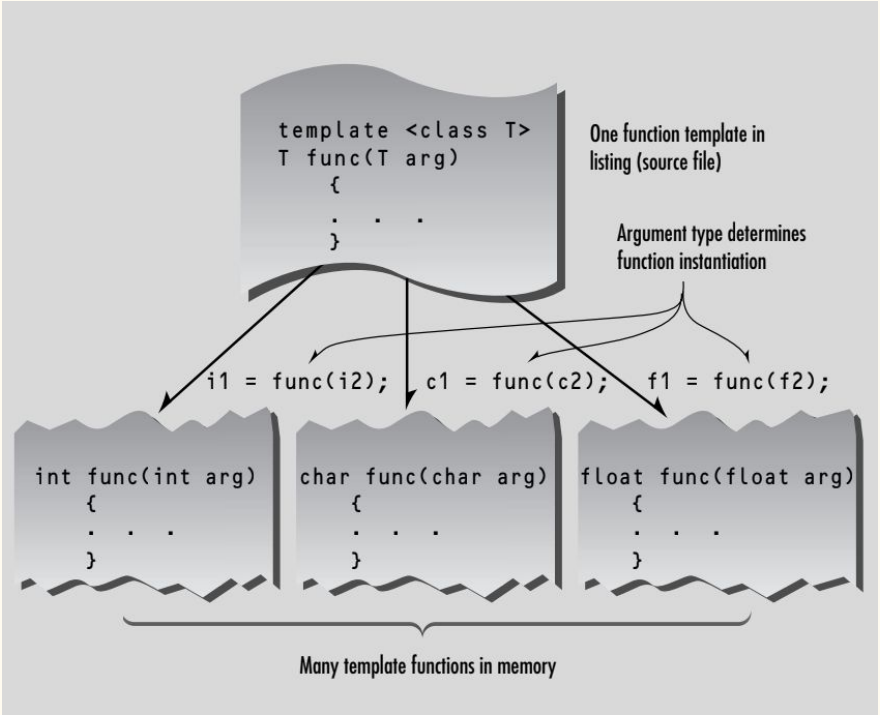
Code generation doesn't take place until the function is actually called (invoked) by a statement within the program.

Instantiating a function template

When the compiler sees a function call, it looks at the type of the argument passed to it, and generates a specific version of the function for this type, substituting the the template argument (T in the previous syntax) in the function template with the appropriate type. This is called *instantiating the function template*.

Each instantiated version of the function is called a template function (i.e., a **template function** is a specific instance of a **function template**.)

Function templates



Function template: Example

```
#include <iostream>

template <typename T> bool greater(T a, T b) {
  return a > b;
}

int main() {
  int a, b;
  std::cout << "Enter two integers: ";
  std::cin >> a >> b;

  std::cout << greater(a, b) << std::endl; // argument type: int
  std::cout << greater<int>(a, b) << std::endl; // argument type: int
  std::cout << "a > b = " << greater<std::string>("a", "b") << std::endl;//arg type: string
}
```

Function templates with multiple arguments

Consider the following function template with multiple arguments:

```
template <class atype>
int find(atype* array, atype value, int size){
  // Function body
}
```

Depending on the arguments passed, the compiler will generate a function

`int find(int*, int, int);` or `int find(float*, float, int);` or `int find(char*, char, int);` or so on.

Function templates with multiple arguments

Consider the following function template with multiple arguments:

```
template <class atype>
int find(atype* array, atype value, int size){
  // Function body
}
```

Note that the compiler expects all instances of `atype` to be the same type. It can generate a function

```
int find(int*, int, int);
```

but it can't generate

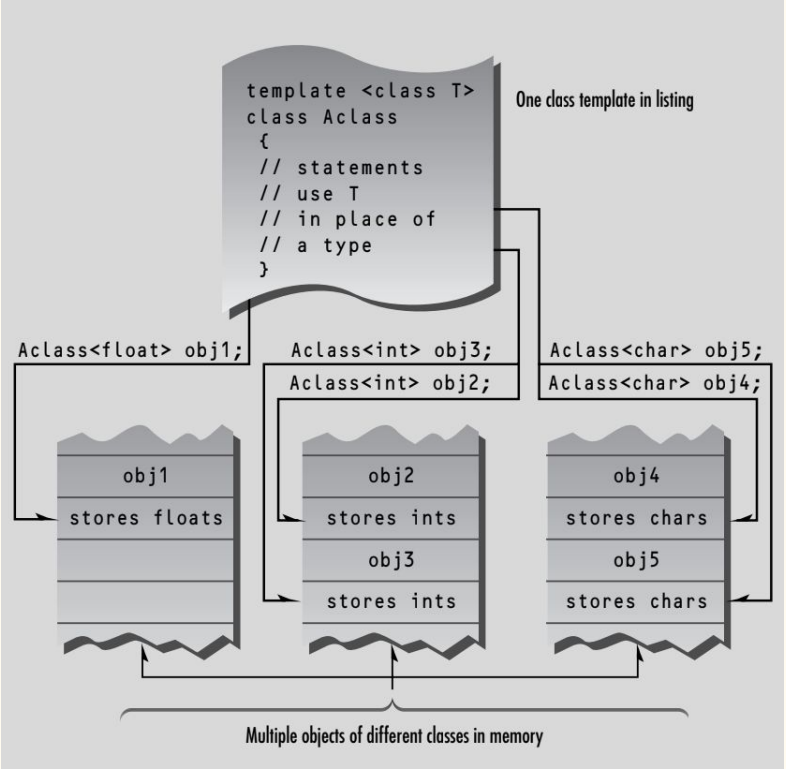
```
int find(int*, float, int);
```

Function template with more than one template argument

```
template <class atype, class btype>
btype find(atype* array, atype value, btype size){
  // Function body
}
```

Class templates

Much like a function template is a template definition for instantiating functions, a class template is a template definition for instantiating class types.



Class templates: Example 1

```cpp
#include <iostream>

template <typename T>
class Pair {
private:
  T first;
  T second;
public:
  Pair(){}
  Pair(T f, T s) : first(f), second(s) {}

  T max() {
    return (first < second ? second : first);
  }
};
```

```cpp
int main()
{
  Pair<int> p1{ 5, 6 }; // Before C++17
  // Pair p1{ 5, 6 }; From C++17
  std::cout << p1.max() << " is larger\n";

  Pair<double> p2{ 1.2, 3.4 };
  std::cout << p2.max() << " is larger\n";

  return 0;
}
```

Class templates: Example 2

```
#include <iostream>

template <typename T>
class Pair {
public:
  T first;
  T second;
  Pair(){}
  Pair(T f, T s) : first(f), second(s) {}
};

template <typename T>
T max(Pair<T> p) {
  return (p.first < p.second ?
            p.second : p.first);
}
```

```
int main()
{
  Pair<int> p1{ 5, 6 };
  std::cout << max(p1) << " is larger\n";

  Pair<double> p2{ 1.2, 3.4 };
  std::cout << max(p2) << " is larger\n";

  return 0;
}
```

Lab 7

Question 1: Implement bubble sort algorithm using function templates.

Question 2: Convert IQueue and ArrayQueue classes (that you developed in Lab 6) into generic classes to enable them to work with any type of data.

Assignment 4

1. What is Virtual function? Why do we need Virtual function?
2. What is Virtual Destructor? Explain how Virtual Destructor avoids memory leakage in the case of Inheritance.
3. Differentiate between Interface class and Virtual Base class?
4. Why do we need to handle exceptions? What is the mechanism in C++ to handle it?
5. What do you mean by Generic Programming? Explain Function Template and Class Template.

References

1. https://www.learncpp.com/cpp-tutorial/class-templates/
2. https://www.learncpp.com/cpp-tutorial/function-templates/
3. Lafore, R. Object Oriented Programming in C++. Sams Publishing.