

Forum:[Hack Forums](#)

Thread: [Templates \(C++\)](#)

World of Warcraft - March 26, 2014, 12:26 am

There are two kinds of templates: function templates and class templates. (A third kind, variable templates, is expected to feature in [C++14](#).)

Function templates

A function template behaves like a function except that the template can have arguments of many different types (see example). In other words, a function template represents a family of functions. The format for declaring function templates with type parameters is

```
template <class identifier> function_declaration;  
template <typename identifier> function_declaration;
```

Both expressions have exactly the same meaning and behave exactly the same way. The latter form was introduced to avoid confusion because a type parameter does not need to be a class, it may also be a basic type like int or double.

For example, the C++ Standard Library contains the function template `max(x, y)` which returns either `x` or `y`, whichever is larger. `max()` could be defined with the following template: //This example throws the following error : call of overloaded 'max(double, double)' is ambiguous

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

This single function definition works with many data types. Although usage of a function template saves space in the source code file (in addition to limiting changes to one function description) versus separate functions written for various datatypes, it does not produce smaller object code than would occur from separate non-templated versions of a function written for different types. For example, if a program uses an int and a double version of the `max()` function template shown above, the compiler will create an object code version of `max()` that takes ints and an object code version that takes doubles. The compiler output will be identical to what would have been produced if the source code contained two non-templated versions of `max()`, one written to handle ints and one written to handle doubles.

```
#include <iostream>
```

```
int main()  
{  
    ??// This will call max<int> (by argument deduction)  
    ??std::cout << max(3, 7) << std::endl;  
    ??// This will call max<double> (by argument deduction)  
    ??std::cout << max(3.0, 7.0) << std::endl;  
    ??// This type is ambiguous, so explicitly instantiate max<double>  
    ??std::cout << max<double>(3, 7.0) << std::endl;  
}
```

```
    ??return 0;  
}
```

In the first two cases, the template argument Type is automatically deduced by the compiler to be int and double, respectively. In the third case deduction fails because the type of the parameters must in general match the template arguments exactly. This function template can be instantiated with any copy-constructible type for which the expression (y < x) is valid. For user-defined types, this implies that the less-than operator must be overloaded.

Class templates

A class template provides a specification for generating classes based on parameters. Class templates are generally used to implement containers. A class template is instantiated by passing a given set of types to it as template arguments. The C++ Standard Library contains many class templates, in particular the containers adapted from the Standard Template Library, such as vector.

Template specialization

When a function or class is instantiated from a template, a specialization of that template is created by the compiler for the set of arguments used, and the specialization is referred to as being a generated specialization.

Explicit template specialization

Sometimes, the programmer may decide to implement a special version of a function (or class) for a given set of template type arguments which is called an explicit specialization. In this way certain template types can have a specialized implementation that is optimized for the type or more meaningful implementation than the generic implementation.

If a class template is specialized by a subset of its parameters it is called partial template specialization (function templates cannot be partially specialized).

If all of the parameters are specialized it is a full specialization.

Explicit specialization is used when the behavior of a function or class for particular choices of the template parameters must deviate from the generic behavior: that is, from the code generated by the main template, or templates. For example, the template definition below defines a specific implementation of template "max" for type "bool":

```
template <>  
bool max<bool>(bool a, bool b) {  
    return a || b;  
}
```

Variadic templates

C++11 introduced variadic templates, which can take a variable number of arguments in a manner somewhat similar to variadic functions such as std::printf. Both function templates and class templates can be variadic.

6577657534735 - March 26, 2014, 12:36 am

Excellent thread, really explained them in more detail than I understood before, but there are a few critiques I feel I must add

I think the vocabulary used in describing these is important because it confused me for a while. They're not really "arguments" in the same way you use the word "arguments" for a function. They are text substitutions similar to a macro and I think this is an important thing to mention.

World of Warcraft - *March 26, 2014, 12:45 am*

Would just like to add that this is from Wikipedia. Yes, took some time to format code and choose good-looking colors.

I will expand on this, make it more clear and cover more. Thanks for your input.