WIKIPEDIA

Template (C++)

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

Templates are of great utility to programmers in C++, especially when combined with multiple inheritance and operator overloading. The C++ Standard Library provides many useful functions within a framework of connected templates.

Major inspirations for C++ templates were the parameterized modules provided by CLU and the generics provided by Ada.[1]

Contents

Technical overview

There are three kinds of templates: *function templates*, *class templates* and, since C++14, *variable templates*. Since C++11, templates may be either variadic or non-variadic; in earlier versions of C++ they are always non-variadic.

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 the same meaning and behave in exactly the same way. The latter form was introduced to avoid confusion,[2] since a type parameter need not be a class. (It can also be a basic type such as `int` or `double`.)

For example, the C++ Standard Library contains the function template `max(x, y)` which returns the larger of `x` and `y`. That function template could be defined like this:

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

This single function definition works with many data types. Specifically, it works with all data types for which > (the greater-than operator) is defined. The usage of a function template saves space in the source code file in addition to limiting changes to one function description and making the code easier to read.

A template does not produce smaller object code, though, compared to writing separate functions for all the different data types used in a specific program. For example, if a program uses both an `int` and a `double` version of the `max()` function template shown above, the compiler will create an object code version of `max()` that operates on `int` arguments and another object code version that operates on `double` arguments. The compiler output will be identical to what would have been produced if the source code had contained two separate non-templated versions of `max()`, one written to handle `int` and one written to handle `double`.

Here is how the function template could be used:

```
#include <iostream>

int main()
{
    // This will call max<int> by implicit argument deduction.
    std::cout << max(3, 7) << std::endl;

    // This will call max<double> by implicit argument deduction.
    std::cout << max(3.0, 7.0) << std::endl;

    // This depends on the compiler. Some compilers handle this by defining a template
    // function like double max <double> ( double a, double b);, while in some compilers
    // we need to explicitly cast it, like std::cout << max<double>(3,7.0);
    std::cout << max(3, 7.0) << std::endl;
    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 automatic deduction of `max(3, 7.0)` would fail because the type of the parameters must in general match the template arguments exactly. Therefore, we explicitly instantiate the `double` version with `max<double>()`.

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 greater-than operator (`>`) must be overloaded in the type.

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.[3] The C++ Standard Library contains many class templates, in particular the containers adapted from the Standard Template Library, such as `vector`.

Variable templates

In C++14, templates can be also used for variables, as in the following example:

```
template<typename T> constexpr T pi = T(3.1415926535897932384626433831L);
```

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 a 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 `max()` for arguments of 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`. Function templates, class templates and (in C++14) variable templates can all be variadic.

Template aliases

C++11 introduced template aliases, which act like parameterized typedefs.

The following code shows the definition of a template alias `StrMap`. This allows, for example, `StrMap<int>` to be used as shorthand for `std::unordered_map<int,std::string>`.

```
template<class T>
using StrMap = std::unordered_map<T, std::string>;
```

Advantages and disadvantages of templates over macros

Some uses of templates, such as the `max()` function mentioned above, were previously fulfilled by function-like preprocessor macros. For example, the following is a C++ `max()` macro that evaluates to the maximum of its two arguments as defined by the < operator:

```
#define max(a,b) ((a) < (b) ? (b) : (a))
```

ie: macros always expand inline, whereas function templates expand into a bunch of function **definitions**, which may or may not be inline!

Both macros and templates are expanded at compile time. Macros are always expanded inline, while templates are only expanded inline when the compiler deems it appropriate. When expanded inline, macro functions and function templates have no extraneous runtime overhead. Template functions with many lines of code will incur runtime overhead when they are not expanded inline, but the reduction in code size may help the code to fit into the CPU's instruction cache.

Macro arguments are not evaluated prior to expansion. The expression using the macro defined above

```
max(0, std::rand() - 100)
```

<-- susceptible to the macro double-evaluation bug!

may evaluate to a negative number (because std::rand() will be called twice as specified in the macro, using different random numbers for comparison and output respectively), while the call to template function

```
std::max(0, std::rand() - 100)
```

will always evaluate to a non-negative number.

As opposed to macros, templates are considered type-safe; that is, they require type-checking at compile time. Hence, the compiler can determine at compile time whether the type associated with a template definition can perform all of the functions required by that template definition.

By design, templates can be utilized in very complex problem spaces, whereas macros are substantially more limited.

There are fundamental drawbacks to the use of templates:

1. Historically, some compilers exhibited poor support for templates. So, the use of templates could decrease code portability.
2. Many compilers lack clear instructions when they detect a template definition error. This can increase the effort of developing templates, and has prompted the development of Concepts for possible inclusion in a future C++ standard.
3. Since the compiler generates additional code for each template type, indiscriminate use of templates can lead to code bloat, resulting in larger executables.
4. Because a template by its nature exposes its implementation, injudicious use in large systems can lead to longer build times.
5. It can be difficult to debug code that is developed using templates. Since the compiler replaces the templates, it becomes difficult for the debugger to locate the code at runtime.
6. Templates of templates (nested templates) are not supported by all compilers, or might have a limit on the nesting level.
7. Templates are in the headers, which require a complete rebuild of all project pieces when changes are made.
8. No information hiding. All code is exposed in the header file. No one library can solely contain the code.

Additionally, the use of the "less than" and "greater than" signs as delimiters is problematic for tools (such as text editors) which analyze source code syntactically. It is difficult for such tools to determine whether a use of these tokens is as comparison operators or template delimiters. For example, this line of code:

```
foo (a < b, c > d) ;
```

may be a function call with two parameters, each the result of a comparison expression. Alternatively, it could be a declaration of a constructor for class `foo` taking a parameter `d` whose type is the parameterized `a < b, c >`.

Generic programming features in other languages

Initially, the concept of templates was not included in some languages, such as Java and C# 1.0. Java's adoption of generics mimics the behavior of templates, but is technically different. C# added generics (parameterized types) in .NET 2.0. The generics in Ada predate C++ templates.

Although C++ templates, Java generics, and .NET generics are often considered similar, generics only mimic the basic behavior of C++ templates.[4] Some of the advanced template features utilized by libraries such as Boost and STLSoft, and implementations of the STL itself, for template metaprogramming (explicit or partial specialization, default template arguments, template non-type arguments, template template arguments, ...) are not available with generics.

In C++ templates, compile-time cases were historically performed by pattern matching over the template arguments. For example, the template base class in the Factorial example below is implemented by matching 0 rather than with an inequality test, which was previously unavailable. However, the arrival in C++11 of standard library features such as std::conditional has provided another, more flexible way to handle conditional template instantiation.

```
// Induction

template <unsigned N>
struct Factorial {
  static const unsigned value = N * Factorial<N - 1>::value;
};

// Base case via template specialization:

template <>
struct Factorial<0> {
  static const unsigned value = 1;
};
```

Recursive `Factorial` template for **compile-time** computation of factorials! BUT...using a `constexpr` function is **much easier** and can also be computed/evaluated at compile-time!

With these definitions, one can compute, say 6! at compile time using the expression `Factorial<6>::value`. Alternatively, constexpr in C++11 can be used to calculate such values directly using a function at compile-time.

See also

- Template metaprogramming
- Metaprogramming
- Generic programming
- Substitution failure is not an error
- Curiously recurring template pattern
- List of C++ template libraries

References

1. Stroustrup, Bjarne (2004-09-08). "The C++ Programming Language (Third Edition and Special Edition)" (http://www.stroustrup.com/). *Bjarne Stroustrup's homepage*.
2. Lippman, Stan. "Why C++ Supports both Class and Typename for Type Parameters" (http://blogs.msdn.com/b/slippman/archive/2004/08/11/212768.aspx). *MSDN*.
3. Vandevoorde, Daveed; Josuttis, Nicolai (2002). *C++ Templates: The Complete Guide*. Addison Wesley. ISBN 978-0-201-73484-3.
4. Differences Between C++ Templates and C# Generics (C# Programming Guide) (http://msdn.microsoft.com/en-us/library/c6cyy67b.aspx)

External links

- Demonstration of the Turing-completeness of C++ templates (http://matt.might.net/articles/c++-template-meta-programming-with-lambda-calculus/) (Lambda calculus implementation)