Covers C++11, C++14, and C++17

C++ Templates

The Complete Guide

SECOND EDITION

David VANDEVOORDE
Nicolai M. JOSUTTIS
Douglas GREGOR

C++ Templates

Second Edition

```
    friend void ::multiply(int);
                        // refers to an instance of the template

    friend void ::multiply<double*>(double*);
                        // qualified names can also have angle brackets,
                        // but a template must be visible

    friend void ::error() { }
                        // ERROR: a qualified friend cannot be a definition
};
```

In our previous examples, we declared the friend functions in an ordinary class. The same rules apply when we declare them in class templates, but the template parameters may participate in identifying the function that is to be a friend:

```
template<typename T>
class Node {
    Node<T>* allocate();
    ...
};

template<typename T>
class List {
    friend Node<T>* Node<T>::allocate();
    ...
};
```

A friend function may also be *defined* within a class template, in which case it is only instantiated when it is actually used. This typically requires the friend function to use the class template itself in the type of the friend function, which makes it easier to express functions on the class template that can be called as if they were visible in namespace scope:

```
template<typename T>
class Creator {
  friend void feed(Creator<T>) {   // every T instantiates a different function ::feed()
    ...
  }
};

int main()
{
  Creator<void> one;
  feed(one);                       // instantiates ::feed(Creator<void>)
  Creator<double> two;
  feed(two);                       // instantiates ::feed(Creator<double>)
}
```

In this example, every instantiation of `Creator` generates a different function. Note that even though these functions are generated as part of the instantiation of a template, the functions themselves are ordinary functions, not instances of a template. However, they are considered *templated entities* (see Section 12.1 on page 181) and their definition is instantiated only when used. Also note that because the body of these functions is defined inside a class definition, they are implicitly inline. Hence, it is not an error for the same function to be generated in two different translation units. Section 13.2.2 on page 220 and Section 21.2.1 on page 497 have more to say about this topic.

12.5.3 Friend Templates

Usually when declaring a friend that is an instance of a function or a class template, we can express exactly which entity is to be the friend. Sometimes it is nonetheless useful to express that all instances of a template are friends of a class. This requires a *friend template*. For example:

```
class Manager {
    template<typename T>
      friend class Task;

    template<typename T>
      friend void Schedule<T>::dispatch(Task<T>*);

    template<typename T>
      friend int ticket() {
          return ++Manager::counter;
      }
    static int counter;
};
```

Just as with ordinary friend declarations, a friend template can be a definition only if it names an unqualified function name that is not followed by angle brackets.

A friend template can declare only primary templates and members of primary templates. Any partial specializations and explicit specializations associated with a primary template are automatically considered friends too.

12.6 Afternotes

The general concept and syntax of C++ templates have remained relatively stable since their inception in the late 1980s. Class templates and function templates were part of the initial template facility. So were type parameters and nontype parameters.

However, there were also some significant additions to the original design, mostly driven by the needs of the C++ standard library. Member templates may well be the most fundamental of those additions. Curiously, only member *function* templates were formally voted into the C++ standard. Member *class* templates became part of the standard by an editorial oversight.

Friend templates, default template arguments, and template template parameters came afterward during the standardization of C++98. The ability to declare template template parameters is sometimes called *higher-order genericity*. They were originally introduced to support a certain allocator model in the C++ standard library, but that allocator model was later replaced by one that does not rely on template template parameters. Later, template template parameters came close to being removed from the language because their specification had remained incomplete until very late in the standardization process for the 1998 standard. Eventually a majority of committee members voted to keep them and their specifications were completed.

Alias templates were introduced as part of the 2011 standard. Alias templates serve the same needs as the oft-requested "typedef templates" feature by making it easy to write a template that is merely a different spelling of an existing class template. The specification (N2258) that made it into the standard was authored by Gabriel Dos Reis and Bjarne Stroustrup; Mat Marcus also contributed to some of the early drafts of that proposal. Gaby also worked out the details of the variable template proposal for C++14 (N3651). Originally, the proposal only intended to support `constexpr` variables, but that restriction was lifted by the time it was adopted in the draft standard.

Variadic templates were driven by the needs of the C++11 standard library and the Boost libraries (see [*Boost*]), where C++ template libraries were using increasingly advanced (and convoluted) techniques to provide templates that accept an arbitrary number of template arguments. Doug Gregor, Jaakko Järvi, Gary Powell, Jens Maurer, and Jason Merrill provided the initial specification for the standard (N2242). Doug also developed the original implementation of the feature (in GNU's GCC) while the specification was being developed, which much helped the ability to use the feature in the standard library.

Fold expressions were the work of Andrew Sutton and Richard Smith: They were added to C++17 through their paper N4191.

Chapter 13

Names in Templates

Names are a fundamental concept in most programming languages. They are the means by which a programmer can refer to previously constructed entities. When a C++ compiler encounters a name, it must "look it up" to identify the entity being referred. From an implementer's point of view, C++ is a hard language in this respect. Consider the C++ statement `x*y;`. If `x` and `y` are the names of variables, this statement is a multiplication, but if `x` is the name of a type, then the statement declares `y` as a pointer to an entity of type `x`.

This small example demonstrates that C++ (like C) is a *context-sensitive language*: A construct cannot always be understood without knowing its wider context. How does this relate to templates? Well, templates are constructs that must deal with multiple wider contexts: (1) the context in which the template appears, (2) the context in which the template is instantiated, and (3) the contexts associated with the template arguments for which the template is instantiated. Hence it should not be totally surprising that "names" must be dealt with quite carefully in C++.

13.1 Name Taxonomy

C++ classifies names in a variety of ways—a large variety of ways, in fact. To help cope with this abundance of terminology, we provide Table 13.1 and Table 13.2, which describe these classifications. Fortunately, you can gain good insight into most C++ template issues by familiarizing yourself with two major naming concepts:

1. A name is a *qualified name* if the scope to which it belongs is explicitly denoted using a scope-resolution operator (`::`) or a member access operator (`.` or `->`). For example, `this->count` is a qualified name, but `count` is not (even though the plain `count` might actually refer to a class member).
2. A name is a *dependent name* if it depends in some way on a template parameter. For example, `std::vector<T>::iterator` is usually a dependent name if `T` is a template parameter, but it is a nondependent name if `T` is a known type alias (such as the `T` from `using T = int`).

Classification	Explanation and Notes
Identifier	A name that consists solely of an uninterrupted sequences of letters, underscores (_), and digits. It cannot start with a digit, and some identifiers are reserved for the implementation: You should not introduce them in your programs (as a rule of thumb, avoid leading underscores and double underscores). The concept of "letter" should be taken broadly and includes special *universal character names* (*UCNs*) that encode glyphs from nonalphabetical languages.
Operator-function-id	The keyword `operator` followed by the symbol for an operator—for example, `operator new` and `operator []`.[1]
Conversion-function-id	Used to denote a user-defined implicit conversion operator—for example, `operator int&`, which could also be obfuscated as `operator int bitand`.
Literal-operator-id	Used to denote a user-defined literal operator—for example, `operator ""_km`, which will be used when writing a literal such as `100_km` (introduced in C++11).
Template-id	The name of a template followed by template arguments enclosed in angle brackets; for example, `List<T, int, 0>`. A template-id may also be an operator-function-id or a literal-operator-id followed by template arguments enclosed in angle brackets; for example, `operator+<X<int>>`.
Unqualified-id	The generalization of an identifier. It can be any of the above (identifier, operator-function-id, conversion-function-id, literal-operator-id, or template-id) or a "destructor name" (e.g., notations like `~Data` or `~List<T, T, N>`).
Qualified-id	An unqualified-id that is qualified with the name of a class, enum, or namespace, or just with the global scope resolution operator. Note that such a name itself can be qualified. Examples are `::X`, `S::x`, `Array<T>::y`, and `::N::A<T>::z`.
Qualified name	This term is not defined in the standard, but we use it to refer to names that undergo *qualified lookup*. Specifically, this is a qualified-id or an unqualified-id that is used after an explicit member access operator (`.` or `->`). Examples are `S::x`, `this->f`, and `p->A::m`. However, just `class_mem` in a context that is implicitly equivalent to `this->class_mem` is not a qualified name: The member access must be explicit.
Unqualified name	An unqualified-id that is not a qualified name. This is not a standard term but corresponds to names that undergo what the standard calls *unqualified lookup*.
Name	Either a qualified or an unqualified name.

Table 13.1. Name Taxonomy (Part 1)

Classification	Explanation and Notes
Dependent name	A name that depends in some way on a template parameter. Typically, a qualified or unqualified name that explicitly contains a template parameter is dependent. Furthermore, a qualified name that is qualified by a member access operator (`.` or `->`) is typically dependent if the type of the expression on the left of the access operator is *type-dependent*, a concept that is discussed in Section 13.3.6 on page 233. In particular, `b` in `this->b` is generally a dependent name when it appears in a template. Finally, a name that is subject to argument-dependent lookup (described in Section 13.2 on page 217), such as `ident` in a call of the form `ident(x, y)` or `+` in the expression `x + y`, is a dependent name if and only if any of the argument expressions is type-dependent.
Nondependent name	A name that is not a dependent name by the above description.

Table 13.2. Name Taxonomy (Part 2)

It is useful to read through the tables to gain some familiarity with the terms that are sometimes used to describe C++ template issues, but it is not essential to remember the exact meaning of every term. Should the need arise, they can be found easily in the index.

13.2 Looking Up Names

There are many small details to looking up names in C++, but we will focus only on a few major concepts. The details are necessary to ensure only that (1) normal cases are treated intuitively, and (2) pathological cases are covered in some way by the standard.

Qualified names are looked up in the scope implied by the qualifying construct. If that scope is a class, then base classes may also be searched. However, enclosing scopes are not considered when looking up qualified names. The following illustrates this basic principle:

```
int x;

class B {
  public:
    int i;
};

class D : public B {
};

void f(D* pd)
{
    pd->i = 3;  // finds B::i
    D::x = 2;   // ERROR: does not find ::x in the enclosing scope
}
```