THE EXPERT'S VOICE® IN C++

Advanced Metaprogramming in Classic C++

Davide Di Gennaro

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

PART 1

#include <prerequisites>
#include <techniques>
#include <applications>

CHAPTER 1

Templates

"C++ supports a variety of styles."

Bjarne Stroustrup, A Perspective on ISO C++

Programming is the process of teaching something to a computer by talking to the machine in one of its common languages. The closer to the machine idiom you go, the less natural the words become.

Each language carries its own expressive power. For any given concept, there is a language where its description is simpler, more concise, and more detailed. In assembler, we have to give an extremely rich and precise description for any (possibly simple) algorithm, and this makes it very hard to read back. On the other hand, the beauty of C++ is that, while being close enough to the machine language, the language carries enough instruments to enrich itself.

C++ allows programmers to express the same concept with different *styles* and good C++ looks more natural.

First you are going to see the connection between the templates and the style, and then you will dig into the details of the C++ template system.

Given this C++ fragment:

```
double x = sq(3.14);
```

Can you guess what `sq` is? It could be a macro:

```
#define sq(x) ((x)*(x))
```

A function:

```
double sq(double x)
{
   return x*x;
}
```

A function template:

```
template <typename scalar_t>
inline scalar_t sq(const scalar_t& x)
{
   return x*x;
}
```

A type (an unnamed instance of a class that decays to a double):

```
class sq
{
   double s_;

public:

   sq(double x)
   : s_(x*x)
   {}

   operator double() const
   { return s_; }
};
```

A global object:

```
class sq_t
{
public:
   typedef double value_type;

   value_type operator()(double x) const
   {
      return x*x;
   }
};

const sq_t sq = sq_t();
```

Regardless of how `sq(3.14)` is implemented, most humans can guess what `sq(3.14)` does just looking at it. However, *visual equivalence* does not imply *interchangeableness*. If `sq` is a class, for example, passing a square to a function template will trigger an unexpected argument deduction:

```
template <typename T> void f(T x);

f(cos(3.14)); // instantiates f<double>
f(sq(3.14));  // instantiates f<sq>. counterintuitive?
```

Furthermore, you would expect every possible numeric type to be squared as efficiently as possible, but different implementations may perform differently in different situations:

```
std::vector<double> v;
std::transform(v.begin(), v.end(), v.begin(), sq);
```

If you need to transform a sequence, most compilers will get a performance boost from the last implementation of `sq` (and an error if `sq` is a macro).

The purpose of TMP is to write code that is:

- Visually clear to human users so that nobody needs to look underneath.
- Efficient in most/all situations from the point of view of the compiler.
- Self-adapting to the rest of the program.[1]

Self-adapting means "portable" (independent of any particular compiler) and "not imposing constraints". An implementation of `sq` that requires its argument to derive from some abstract base class would not qualify as self-adapting.

The true power of C++ templates is *style*. Compare the following equivalent lines:

```
double x1 = (-b + sqrt(b*b-4*a*c))/(2*a);

double x2 = (-b + sqrt(sq(b)-4*a*c))/(2*a);
```

All template argument computations and deductions are performed at compile time, so they impose no runtime overhead. If the function `sq` is properly written, line 2 is at least as efficient as line 1 and easier to read at the same time.

Using `sq` is elegant:

- It makes code readable or self-evident
- It carries no speed penalty
- It leaves the program open to future optimizations

In fact, after the concept of squaring has been isolated from plain multiplication, you can easily plug in specializations:

```
template <typename scalar_t>
inline scalar_t sq(const scalar_t& x)
{
   return x*x;
}

template <>
inline double sq(const double& x)
{
   // here, use any special algorithm you have!
}
```

[1]Loosely speaking, that's the reason for the "meta" prefix in "metaprogramming".

1.1. C++ Templates

The classic C++ language admits two basic types of templates—*function templates* and *class templates*[2]:

Here is a function template:

```
template <typename scalar_t>
scalar_t sq(const scalar_t& x)
{
   return x*x;
}
```

Here is a class template:

```
template
<
   typename scalar_t,              // type parameter
   bool EXTRA_PRECISION = false,   // bool parameter with default value
   typename promotion_t = scalar_t // type parameter with default value
>
class sum
{
   // ...
};
```

When you supply suitable values to all its parameters, a template generates entities during compilation. A function template will produce functions and a class template will produce classes. The most important ideas from the TMP viewpoint can be summarized as follows:

- You can exploit class templates to perform computations at compile time.
- Function templates can auto-deduce their parameters from arguments. If you call `sq(3.14)`, the compiler will automatically figure out that `scalar_t` is `double`, generate the function `sq<double>`, and insert it at the call site.

Both kinds of template entities start declaring a *parameter list* in angle brackets. Parameters can include *types* (declared with the keyword `typename` or `class`) and non-types: integers and pointers.[3]

Note that, when the parameter list is long or when you simply want to comment each parameter separately, you may want to indent it as if it were a block of code within curly brackets.

Parameters can in fact have a default value:

```
sum<double> S1;                 // template argument is 'double', EXTRA_PRECISION is false
sum<double, true> S2;
```

[2]In modern C++ there are more, but you can consider them extensions; the ones described here are metaprogramming first-class citizens. Chapter 12 has more details.

[3]Usually any integer type is accepted, including named/anonymous `enum`, `bool`, typedefs (like `ptrdiff_t` and `size_t`), and even compiler-specific types (for example, `__int64` in MSVC). Pointers to member/global functions are allowed with no restriction; a pointer to a variable (having external linkage) is legal, but it cannot be dereferenced *at compile time*, so this has very limited use in practice. See Chapter 11.

A template can be seen as a metafunction that maps a tuple of parameters to a function or a class. For example, the `sq` template

```
template <typename scalar_t>
scalar_t sq(const scalar_t& x);
```

maps a type `T` to a function:

```
T → T (*)(const T&)
```

In other words, `sq<double>` is a function with signature `double (*)(const double&)`. Note that `double` is the value of the parameter `scalar_t`.

Conversely, the class template

```
template <typename char_t = char>
class basic_string;
```

maps a type `T` to a class:

```
T → basic_string<T>
```

With classes, *explicit specialization* can limit the domain of the metafunction. You have a general template and then some specializations; each of these may or may not have a body.

```
// the following template can be instantiated
// only on char and wchar_t

template <typename char_t = char>
class basic_string;
// note: no body

template < >
class basic_string<char>
{ ... };

template < >
class basic_string<wchar_t>
{ ... };
```

`char_t` and `scalar_t` are called *template parameters*. When `basic_string<char>` and `sq<double>` are used, `char` and `double` are called *template arguments*, even if there may be some confusion between `double` (the template argument of `sq`) and `x` (the argument of the function `sq<double>`).

When you supply template arguments (both types and non-types) to the template, seen as a metafunction, the template is *instantiated*, so if necessary the compiler produces machine code for the entity that the template produces.

Note that different arguments yield different instances, even when instances themselves are identical: `sq<double>` and `sq<const double>` are two unrelated functions.[4]

[4]The linker may eventually collapse them, as they will likely produce identical machine code, but from a language perspective they are different.

When using function templates, the compiler will usually figure out the parameters. We say that an argument *binds* to a template parameter.

```
template <typename scalar_t>
scalar_t sq(const scalar_t& x) { return x*x; }

double pi = 3.14;
sq(pi);                          // the compiler "binds" double to scalar_t

double x = sq(3.14);             // ok: the compiler deduces that scalar_t is double
double x = sq<double>(3.14);     // this is legal, but less than ideal
```

All template arguments must be compile-time constants.

- Type parameters will accept everything known to be a type.
- Non-type parameters work according to the most automatic casting/promotion rule.[5]

Here are some typical errors:

```
template <int N>
class SomeClass
{
};

int main()
{
   int A = rand();
   SomeClass<A> s;      // error: A is not a compile time constant

   const int B = rand();
   SomeClass<B> s;      // error: B is not a compile time constant

   static const int C = 2;
   SomeClass<C> s;      // OK

}
```

The best syntax for a compile-time constant in classic C++ is `static const [[integer type]] name = value;`.

The static prefix could be omitted if the constant is local, in the body of a function, as shown previously. However, it's both harmless and clear (you can find all the compile-time constants in a project by searching for "`static const`" rather than "`const`" alone).[6]

[5]An exception being that literal 0 may not be a valid pointer.
[6]See Sections 1.3.6 and **11.2.2** for more complete discussions.

The arguments passed to the template can be the result of a (compile-time) computation. Every valid integer operation can be evaluated on compile-time constants:

- Division by zero causes a compiler error.
- Function calls are forbidden.[7]
- Code that produces an intermediate object of non-integer/non-pointer type is non-portable, except when inside `sizeof`: `(int)(N*1.2)`, which is illegal. Instead use `(N+N/5)`. `static_cast<void*>(0)` is fine too.[8]

```
SomeClass<(27+56*5) % 4> s1;
SomeClass<sizeof(void*)*CHAR_BIT> s1;
```

Division by zero will cause a compiler error only if the computation is entirely static. To see the difference, note that this program compiles (but it won't run).

```
template <int N>
struct tricky
{
   int f(int i = 0)
   {
      return i/N;          // i/N is not a constant
   }
};

int main()
{
   tricky<0> t;
   return t.f();
}

test.cpp(5) : warning C4723: potential divide by 0
```

On the other hand, compare the preceding listing with the following two, where the division by zero happens during compilation (in two different contexts):

```
int f()
{
   return N/N;          // N/N is a constant
}

test.cpp(5) : error C2124: divide or mod by zero
        .\test.cpp(5) : while compiling class template member function
                        'int tricky<N>::f(void)'
        with
        [
            N=0
        ]
```

[7]See the note in Section 1.3.2.
[8]You can cast a floating-point literal to integer, so strictly speaking, `(int)(1.2)` is allowed. Not all compilers are rigorous in regard to this rule.

And with:

```
tricky<0/0> t;

test.cpp(12) : error C2975: 'N' : invalid template argument for 'tricky',
expected compile-time constant expression
```

More precisely, compile-time constants can be:

- Integer literals, for example, `27`, `CHAR_BIT`, and `0x05`
- `sizeof` and similar non-standard language operators with an integer result (for example, `__alignof__` where present)
- Non-type template parameters (in the context of an "outer" template)

  ```
  template <int N>
  class AnotherClass
  {
     SomeClass<N> myMember_;
  };
  ```

- Static constants of integer type

  ```
  template <int N, int K>
  struct MyTemplate
  {
     static const int PRODUCT = N*K;
  };

  SomeClass< MyTemplate<10,12>::PRODUCT > s1;
  ```

- Some standard macros, such as `__LINE__` (There is actually some degree of freedom; as a rule they are constants with type `long`, except in implementation-dependent "edit and continue" debug builds, where the compiler must use references. In this case, using the macro will cause a compilation error.)[9]

  ```
  SomeClass<__LINE__> s1;  // usually works...
  ```

A parameter can depend on a previous parameter:

```
template
<
   typename T,
   int (*FUNC)(T)    // pointer to function taking T and returning int
>
class X
{
};
```

[9]The use of `__LINE__` as a parameter in practice occurs rarely; it's popular in automatic type enumerations (see Section 7.6) and in some implementation of custom assertions.

```
template
<
   typename T,     // here the compiler learns that 'T' is a type
   T VALUE         // may be ok or not... the compiler assumes the best
>
class Y
{
};

Y<int, 7> y1;      // fine
Y<double, 3> y2;   // error: the constant '3' cannot have type 'double'
```

Classes (and class templates) may also have *template member functions*:

```
// normal class with template member function

struct mathematics
{
   template <typename scalar_t>
   scalar_t sq(scalar_t x) const
   {
      return x*x;
   }
};

// class template with template member function

template <typename scalar_t>
struct more_mathematics
{
   template <typename other_t>[10]
   static scalar_t product(scalar_t x, other_t y)
   {
      return x*y;
   }
};

double A = mathematics().sq(3.14);
double B = more_mathematics<double>().product(3.14, 5);
```

1.1.1. Typename

The keyword `typename` is used:

- As a synonym of class, when declaring a type template parameter
- Whenever it's not evident to the compiler that an identifier is a type name

[10]We have to choose a different name, to avoid shadowing the outer template parameter `scalar_t`.

For an example of "not evident" think about `MyClass<T>::Y` in the following fragment:

```
template <typename T>
struct MyClass
{
   typedef double Y;                    // Y may or may not be a type
   typedef T Type;                      // Type is always a type
};

template < >
struct MyClass<int>
{
   static const int Y = 314;            // Y may or may not be a type
   typedef int Type;                    // Type is always a type
};

int Q = 8;

template <typename T>
void SomeFunc()
{
  MyClass<T>::Y * Q; // what is this line? it may be:
                     // the declaration of local pointer-to-double named Q;
                     // or the product of the constant 314, times the global variable Q
};
```

`Y` is a *dependent name*, since its meaning depends on `T`, which is an unknown parameter.

Everything that depends directly or indirectly on unknown template parameters is a dependent name. If a dependent name refers to a type, then it must be introduced with the `typename` keyword.

```
template <typename X>
class AnotherClass
{
   MyClass<X>::Type t1_;           // error: 'Type' is a dependent name
   typename MyClass<X>::Type t2_;  // ok

   MyClass<double>::Type t3_;      // ok: 'Type' is independent of X
};
```

Note that `typename` is required in the first case and forbidden in the last:

```
template <typename X>
class AnotherClass
{
   typename MyClass<X>::Y member1_;      // ok, but it won't compile if X is 'int'.
   typename MyClass<double>::Y member2_; // error
};
```

typename may introduce a dependent type when declaring a non-type template parameter:

```
template <typename T, typename T::type N>
struct SomeClass
{
};

struct S1
{
   typedef int type;
};

SomeClass<S1, 3> x;   // ok: N=3 has type 'int'
```

As a curiosity, the classic C++ standard specifies that if the syntax typename T1::T2 yields a non-type during instantiation, then the program is ill-formed. However, it doesn't specify the converse: if T1::T2 has a valid meaning as a non-type, then it could be re-interpreted later as a type, if necessary. For example:

```
template <typename T>
struct B
{
   static const int N = sizeof(A<T>::X);
   // should be: sizeof(typename A...)
};
```

Until instantiation, B "thinks" it's going to call sizeof on a non-type; in particular, sizeof is a valid operator on non-types, so the code is legal. However, X could later resolve to a type, and the code would be legal anyway:

```
template <typename T>
struct A
{
   static const int X = 7;
};

template <>
struct A<char>
{
   typedef double X;
};
```

Although the intent of typename is to forbid all such ambiguities, it may not cover all corner cases.[11]

[11]See also http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#666.

1.1.2. Angle Brackets

Even if all parameters have a default value, you cannot entirely omit the angle brackets:

```
template <typename T = double>
class sum {};

sum<> S1;    // ok, using double
sum S2;      // error
```

Template parameters may carry different meanings:

- Sometimes they are really meant to be generic, for example, `std::vector<T>` or `std::set<T>`. There may be some conceptual assumptions about `T`—say constructible, comparable...—that do not compromise the generality.
- Sometimes parameters are assumed to belong to a fixed set. In this case, the class template is simply the common implementation for two or more similar classes.[12]

In the latter case, you may want to provide a set of regular classes that are used without angle brackets, so you can either derive them from a template base or just use `typedef`[13]:

```
template <typename char_t = char>
class basic_string
{
    // this code compiles only when char_t is either 'char' or 'wchar_t'
    // ...
};

class my_string : public basic_string<>
{
   // empty or minimal body
   // note: no virtual destructor!
};

typedef basic_string<wchar_t> your_string;
```

A popular compiler extension (officially part of C++0x) is that two or more adjacent "close angle brackets" will be parsed as "end of template," not as an "extraction operator". Anyway, with older compilers, it's good practice to add extra spaces:

```
std::vector<std::list<double>> v1;
//                           ^^
// may be parsed as "operator>>"

std::vector<std::list<double> > v2;
//                           ^^^
// always ok
```

[12] Even if it's not a correct example, an open-minded reader may want to consider the relationship between `std::string`, `std::wstring`, and `std::basic_string<T>`.
[13] See 1.4.9.

1.1.3. Universal Constructors

A template copy constructor and an assignment are not called when dealing with two objects of the very same kind:

```
template <typename T>
class something
{
public:
   // not called when S == T
   template <typename S>
   something(const something<S>& that)
   {
   }

   // not called when S == T
   template <typename S>
   something& operator=(const something<S>& that)
   {
      return *this;
   }
};

   something<int> s0;
   something<double> s1, s2;

   s0 = s1;    // calls user defined operator=
   s1 = s2;    // calls the compiler generated assignment
```

The user-defined template members are sometimes called *universal copy constructors* and *universal assignments*. Note that universal operators take `something<X>`, not `X`.

The C++ Standard 12.8 says:

- "Because a template constructor is never a copy constructor, the presence of such a template does not suppress the implicit declaration of a copy constructor."
- "Template constructors participate in overload resolution with other constructors, including copy constructors, and a template constructor may be used to copy an object if it provides a better match than other constructors."

In fact, having very generic template operators in base classes can introduce bugs, as this example shows:

```
struct base
{
   base() {}

   template <typename T>
   base(T x) {}
};
```

```
struct derived : base
{
   derived() {}

   derived(const derived& that)
   : base(that) {}
};

   derived d1;
   derived d2 = d1;
```

The assignment `d2 = d1` causes a stack overflow.

An implicit copy constructor must invoke the copy constructor of the base class, so by 12.8 above it can never call the universal constructor. Had the compiler generated a copy constructor for `derived`, it would have called the `base` copy constructor (which is implicit). Unfortunately, a copy constructor for `derived` is given, and it contains an explicit function call, namely `base(that)`. Hence, following the usual overload resolution rules, it matches the universal constructor with `T=derived`. Since this function takes `x` by value, it needs to perform a copy of `that`, and hence the call is recursive.[14]

1.1.4. Function Types and Function Pointers

Mind the difference between a function type and a pointer-to-function type:

```
template <double F(int)>
struct A
{
};

template <double (*F)(int)>
struct B
{
};
```

They are mostly equivalent:

```
double f(int)
{
   return 3.14;
}

   A<f> t1;   // ok
   B<f> t2;   // ok
```

[14] As a side note, this shows once more that in TMP, the less code you write, the better.

Usually a function decays to a function pointer exactly as an array decays to a pointer. But a function type cannot be constructed, so it will cause failures in code that look harmless:

```
template <typename T>
struct X
{
   T member_;

   X(T value)
   : member_(value)
   {
   }
};

X<double (int)> t1(f);         // error: cannot construct 'member_'
X<double (*)(int)> t2(f);      // ok: 'member_' is a pointer
```

This problem is mostly evident in functions that return a functor (the reader can think about `std::not1` or see Section **4.3.4**). In C++, function templates that get parameters by reference prevent the decay:

```
template <typename T>
X<T> identify_by_val(T x)
{
   return X<T>(x);
}

template <typename T>
X<T> identify_by_ref(const T& x)
{
   return X<T>(x);
}

double f(int)
{
   return 3.14;
}

identify_by_val(f);  // function decays to pointer-to-function:
                     // template instantiated with T = double (*)(int)

identify_by_ref(f);  // no decay:
                     // template instantiated with T = double (int)
```

For what concerns pointers, function templates with explicit parameters behave like ordinary functions:

```
double f(double x)
{
   return x+1;
}
```

```
template <typename T>
T g(T x)
{
   return x+1;
}

typedef double (*FUNC_T)(double);

FUNC_T f1 = f;
FUNC_T f2 = g<double>;
```

However, if they are members of class templates and their context depends on a yet unspecified parameter, they require an extra `template` keyword before their name[15]:

```
template <typename X>
struct outer
{
   template <typename T>
   static T g(T x)
   {
      return x+1;
   }
};

template <typename X>
void do_it()
{
   FUNC_T f1 = outer<X>::g<double>;               // error!
   FUNC_T f2 = outer<X>::template g<double>;      // correct
}
```

Both `typename` and `template` are required for inner template classes:

```
template <typename X>
struct outer
{
   template <typename T>
   struct inner {};
};

template <typename X>
void do_it()
{
   typename outer<X>::template inner<double> I;
}
```

Some compilers are not rigorous at this.

[15]Compare with the use of `typename` described in Section1.1.1.

1.1.5. Non-Template Base Classes

If a class template has members that do not depend on its parameters, it may be convenient to move them into a plain class:

```
template <typename T>
class MyClass
{
   double value_;
   std::string name_;
   std::vector<T> data_;

public:
   std::string getName() const;
};
```

should become:

```
class MyBaseClass
{
protected:
   ~MyBaseClass() {}

   double value_;
   std::string name_;

public:
   std::string getName() const;
};

template <typename T>
class MyClass : MyBaseClass
{
   std::vector<T> data_;

public:
   using MyBaseClass::getName;
};
```

The derivation may be public, private, or even protected.[16] This will reduce the compilation complexity and potentially the size of the binary code. Of course, this optimization is most effective if the template is instantiated many times.

[16]See the "brittle base class problem" mentioned by Bjarne Stroustrup in his "C++ Style and Technique FAQ" at http://www.research.att.com/~bs/.

1.1.6. Template Position

The body of a class/function template must be available to the compiler at every point of instantiation, so the usual `header/cpp` file separation does not hold, and everything is packaged in a single file, with the `hpp` extension.

If only a declaration is available, the compiler will use it, but the linker will return errors:

```
// sq.h

template <typename T>
T sq(const T& x);

// sq.cpp

template <typename T>
T sq(const T& x)
{
   return x*x;
}

// main.cpp

#include "sq.h"            // note: function body not visible

int main()
{
   double x = sq(3.14);    // compiles but does not link
```

A separate header file is useful if you want to publish only some instantiations of the template. For example, the author of `sq` might want to distribute binary files with the code for `sq<int>` and `sq<double>`, so that they are the only valid types.

In C++, it's possible to explicitly force the instantiation of a template entity in a translation unit without ever using it. This is accomplished with the special syntax:

```
template class X<double>;

template double sq<double>(const double&);
```

Adding this line to `sq.cpp` will "export" `sq<double>` as if it were an ordinary function, and the plain inclusion of `sq.h` will suffice to build the program.

This feature is often used with algorithm tags. Suppose you have a function template, say `encrypt` or `compress`, whose algorithmic details must be kept confidential. Template parameter `T` represents an option from a small set (say `T=fast, normal, best`); obviously, users of the algorithm are not supposed to add their own options, so you can force the instantiation of a small number of instances—`encrypt<fast>`, `encrypt<normal>`, and `encrypt<best>`—and distribute just a header and a binary file.

■ **Note** C++0x adds to the language the external instantiation of templates. If the keyword `extern` is used before `template`, the compiler will skip instantiation and the linker will borrow the template body from another translation unit.

See also Section 1.6.1 below.

1.2. Specialization and Argument Deduction

By definition, we say that a name is *at namespace level*, at *class level,* or *at body level* when the name appears between the curly brackets of a namespace, class, or function body, as the following example shows:

```
class X                          // here, X is at namespace level
{
public:
   typedef double value_type;    // value_type is at class level

   X(const X& y)                 // both X and y are at class level
   {
   }

   void f()                      // f is at class level
   {
      int z = 0;                 //   body level
      struct LOCAL {};           //   LOCAL is a local class
   }
};
```

Function templates—member or non-member—can automatically deduce the template argument looking at their argument list. Roughly speaking,[17] the compiler will pick the most specialized function that matches the arguments. An exact match, if feasible, is always preferred, but a conversion can occur.

A function F is more specialized than G if you can replace any call to F with a call to G (on the same arguments), but not vice versa. In addition, a non-template function is considered more specialized than a template with the same name.

Sometimes *overload* and *specialization* look very similar:

```
template <typename scalar_t>
inline scalar_t sq(const scalar_t& x);  // (1) function template

inline double sq(const double& x);      // (2) overload

template <>
inline int sq(const int& x);            // (3) specialization of 1
```

But they are not identical; consider the following counter-example:

```
inline double sq(float x);          // ok, overloaded sq may
                                    // have different signature

template <>                         // error: invalid specialization
inline int sq(const int x);         // it must have the same signature
```

[17]The exact rules are documented and explained in **[2]**. You're invited to refer to this book for a detailed explanation of what's summarized here in a few paragraphs.