C++ Templates

Daniel Noland

December 16, 2014

1 Introduction

From the November 2014 working draft of the official `C++` language specification:

> A template defines a family of classes or functions or an alias for a family of types.

There. Done. You now know what a template is.

Just kidding. Template programming is actually a vast subject. You can consider the template features of `C++` to be a complete programming language in their own right. Template programming has standard techniques, idioms, recursive algorithms. . . it goes on and on. I have been working with templates for years, and I still feel like a novice as often as not.

But don't freak out! Basic template use is actually pretty easy. That said, it has its pitfalls. Just like you can know the definition of a template without understanding what a template is, you can use templates without understanding their raison d'être. As with any tool, learning to use it without learning when and why. . . well it is a sure fire recipe for using your tool badly. Thus, I wrote an essay.

2 `C++` and the type system

`C++` is what we call a *strongly typed* language. That is, it requires each variable you use to be of a specific type or nature. A `double` is a completely different animal than a `float`, even if they are logically connected and can often be used interchangeably. If you learned `python` or `JavaScript (ECMAScript)`, or `perl`, or one of many other languages before you picked up `C++` you were likely shocked to discover that you were required to decorate all your variables with little names like `int`, `float`, or `char`. Most novice programmers see this and simply accept it; just one of many things the mysterious "compiler" bitches about when you forget. . . right? Clearly, the fact that I don't need to waste time telling `python` that I'm using a number when I write

```
someNumber = 2
```

makes `python` a better language, right? I will simply say this: *typed variables exist for a damn good reason.*

Consider this simple (illegal) function:

```
f(a, b) {
    return a + b;
}
```

Now, if this function would compile, I could write

```
f(1, 2)
```

and you would expect to get `3`. I could also write

```
f("impending ", "disaster")
```

and you would expect to get `"impending disaster"`. Nifty…my function is doing double duty: adding numbers and concatenating strings. If only my compiler were smart enough to make this work!

Wrong. Well, mostly wrong anyway. This kind of approach has some advantages (clearly), and for some languages and problem sets this approach works just dandy. I use `python` all the time…it solves lots of my problems. Same story for `bash/zsh`, `JavaScript`, *Mathematica*…the list goes on. So why is `C++` strongly typed?

The answer to that becomes more and more obvious the more you know about how your computer actually works (hint: take a computer systems class). But even ignoring all that, would you actually *want* `C++` to be a weakly typed language? What does the type system buy you?

Well, for starters, it catches a whole host of bugs ***at compile time***. When I make a mistake and try to add a number to a string, I want the computer to stop, think, and then complain. Loudly. Yes, it may make perfect sense to add a number to a string. It may also produce complete nonsense. Consider this:

```
f("my number is ", 3)
```

You might want the string `"my number is 3"`. Say the compiler were to silently oblige. But then, you might end up in this situation

```
f("3", 3)
```

For the sake of consistency, the result must be `"33"`, in spite of the fact that you were very likely trying to add `3 + 3`. We could go a step farther and have the compiler automatically generate code which checked the string, and then do addition if the string looked like a number, and concatenation otherwise. Sounds great. Except for the fact that it is *the worst idea ever.* I don't know about you, but I don't think my computer should be wasting its time running regex algorithms on strings when all I really want is to add `3 + 3`. Especially if I am doing a complex calculation where errors in the answer may not be obvious. My point? *The type system is a gift.*

`C++` gives us the chance to catch countless stupid errors before we ever run our code. Ever compare a `signed int` to an `unsigned int`? `C++` gives you a nice little warning. Ever assign an object of type `X` to one of type `Y`? Program did not compile, did it? **The compiler is doing its best to save us from our stupid selves.**

3 The problem with a strongly typed language

So we are sold. All hail types. Hurrah types. Let's just use function overloading and fix this stupid function.

```
int f(int a, int b) {
    return a + b;
}
string f(string a, string b) {
    return a + b;
}
```

Great. Now we can concatenate `string`s and add `int`s. But what about `float`. Oh, we forgot that one.

```
float f(float a, float b) {
    return a + b;
}
```

Now we are done...right? Except for `double`. Gotta have `double`.

```
double f(double a, double b) {
    return a + b;
}
```

There, got em all! Shit, forgot `long double`.

```
long double f(long double a, long double b) {
    return a + b;
}
```

Oh, and those pesky `unsigned` data types. And `long`. And `short`. And `char`. What about `complex`? Oh...you made a `Matrix` class, and you think my function should be able to add them together? Sounds reasonable. Oh, but it is not one matrix class, but a different matrix class for all the primitive data types? You mean `MatrixOfDouble` and `MatrixOfLong` and `MatrixOfFloat` and `MatrixOfComplex` and `MatrixOfKitchenSink` are all separate data types, and we need to support all of them? But lets not forget about other data structures! List, tree, graph? Should we even be able to add all of these? At the very least we are going to need a table to keep track of all the different data types, but that is fine. After all, we all know how to use spreadsheets.

Does it start to feel like this type system is biting us? If not, consider this: you might need to fix a bug in your `f` function? Get ready to fix it in dozens of different places! Sounds like a fun time right? I know I put that revolver somewhere.

STOP! If you ever find yourself approaching a problem this way **you are quite obviously doing it wrong!** Back off the problem for a moment and think. We get silly bugs without types, and an infinity mirror of function overloading with them. Seems like we are doomed. Time to take up pottery?

4 Templates to the rescue

If computers are good at one thing it is book keeping. You may not be able to manage 47,253 different overloads of your `f` function, but your computer just blasts right through that. The problem is not that we can't generate all of those overloads (just copy/paste and search/replace). The problem is that we can't manage the code once we have generated it.

What if there were a way to write our function once in a highly generic way, and then have the computer generate specific versions of that function as demand dictates? Well, that would have a whole host of advantages. The designers of `C++` thought the same thing. Enter the function template:

```
template<typename T>
T f(T a, T b) {
   return a + b;
}
```

Want to add a couple of `int`s? No problem:

```
int a = 2;
int b = 3;
int c = f(a, b);
```

Tada! Oh, you need `double` as well?

```
double a = 2.3;
double b = 3.2;
double c = f(a, b);
```

Still on about that `MatrixOfKitchenSink` class you spent so much time writing?

```
MatrixOfKitchenSink a;
MatrixOfKitchenSink b;
// Whatever you do with a MatrixOfKitchenSink
MatrixOfKitchenSink c = f(a, b);
```

I have just written a dizzyingly infinite kaleidoscope of functions in 4 little lines of `C++`. Look on my works, ye mighty, and despair!

If you have not guessed, `f` is no longer a true function. It is a *template*. When I use `f`, the compiler looks at its arguments (`a` and `b` in this case), considers their types, and *generates* an appropriate function for me *at compile time*. That is, when I call `f(a, b)`, if `a` and `b` are `double`s, the compiler generates a function which looks like this:

```
double f(double a, double b) {
   return a + b;
}
```

Same story for `long` and `MatrixOfKitchenSink` and whatever other class you want to dream up. In a template function **the type of data you are operating on is variable.** In this case I called the type `T`. But it could just as easily have been written like this:

```
template<typename Bobo>
Bobo f(Bobo a, Bobo b) {
   return a + b;
}
```

C++ does not care. As long as you are using a legal variable name all is well.

Now imagine that you get an email from your lovely employer which says that f needs to subtract b from a instead. You only need to change one thing:

```
template<typename T>
T f(T a, T b) {
   return a - b;
}
```

Done. No revolver or pottery lessons needed.

So far we have this:

1. Types are good: they help us catch dumb mistakes.
2. Types are not perfect: we need to deal with a huge zoo of different types.
3. Templates tame that problem: use templates when you want to write a generic function.

Function templates are really handy. They let us write one generic that can then generate as many overloads as are necessary for our problem. Cool. But templates are not limited to just functions. We can also make templated classes.

5 Template classes

Consider my favorite programming problem: the Matrix class.

```
class Matrix {

   private:

      // 2D array of doubles
      double** data;

   public:

      // Default constructor
      Matrix() {
         // default constructor things
      }

      // Destructor
```

```
        ~Matrix() {
            // destructor things
        }

        Matrix add(Matrix X) {
            // code which adds *this to x
        }

        Matrix subtract(Matrix X) {
            // code which subtracts  x from *this
        }

        // Other methods:
        // multiply by scalar
        // multiply by Matrix
        // invert
        // diagonalize
        // ...
};
```

Ok, basic `Matrix` class done. Except now the scientist you are working with says "hey, my new algorithm requires extra numerical precision in the `Matrix` routines, what can you do for me?" You, being an excellent programmer, say "sure, I will just change `double` to `long double`." And you do. But then some other scientist says: "the new code version is 47% slower than the old one! What gives?" Oh…that change to `long double` made her algorithm slower. That's bad. I guess we need two `Matrix` classes. One for `double` and one for `long double`. Worse still, these scientists are doing some crazy high power math, and they don't just need `complex` as they underlying data types, they need support for exotic stuff like quaternions and what have you. Who knows, tomorrow they may want some bizarre group theory based data type you have never even heard of. Each time they come up with something new, the number of things you need to keep track of multiplies again.

See where I am going? We will end up in the same problem we had with the `f` function before: we need to support multiple data types, but that will quickly spiral out of control. Only this is even worse: that one data type changes the mechanics of multiple different member functions.

Templates can save us again. This is actually pretty easy. Behold:

```
template<typename T>
class Matrix {

    private:

        // 2D array of T
        T** data;
    public:

        // Default constructor
        Matrix() {
            // default constructor things
        }

        // Destructor
        ~Matrix() {
            // destructor things
        }

        // Note: we can't return a Matrix.  We must return a Matrix<SomeType>
        // In this case, it makes sense to return a Matrix<T>
        Matrix<T> add(Matrix<T> X) {
            // code which adds *this to x
        }

        Matrix<T> subtract(Matrix<T> X) {
            // code which subtracts x from *this
        }
        // ...
};
```

Just like with the `f` *function template*, the `Matrix` *class template* allows the compiler to generate a new class for me at compile time. To be clear: **`Matrix` is no longer a class! It is a class template.** I can't make a variable of type `Matrix`. Such a type does not even make sense (think about it!). Instead, I must specify which type of data that `T` type variable is going to hold. Fortunately this is quite easy.

Say I want a `Matrix` with underlying data type `double`. I just write

```
Matrix<double> myMatrix;
```

Oh, you want `long double`?

```
Matrix<long double> yourMatrix;
```

This is great! We can take a class template like `Matrix`, and generate a class `Matrix<float>` from it! Go back and consider the relative merits and weaknesses of a strongly typed language. Then think about class templates for a few minutes. This class template business is pretty close to having the best of both worlds. We get the flexibility of a weakly typed language with the *type safety* of a strongly typed one.

6 Baby steps

I guess this section is "optional" in that I have already expressed the basic essence of template programming. Alternatively, you can look at it like this: you are finally playing with a full deck of cards. You have seen, with a few notable exceptions (*cough* inheritance / polymorphism *cough*), most of the major features of C++. You are much much closer to being able to write professional quality code. So we may as well push a little harder and cover a few of the more sophisticated features of template programming.

6.1 Multiple template parameters

What if you need to write a function or class which has multiple variable data types? This actually happens constantly. The most trivial example I can dream up is a class whose single job is to store two different values. The trick is that the values need to be of arbitrary type.[1] Say we call the class template pair:

```
template<typename T, typename U>
class pair {
   public:
      T first;
      U second;
};
```

You might think this class useless, but you would be very wrong. I will save that discussion for the next essay. The point is, you can use as many template parameters as you like. You can even make variadic templates (templates which take a variable number of parameters), but that is outside the scope of this essay.

If you wanted to actually use this pair template it would look something like this:

```
pair<double, string> somePair;
somePair.first = 3.2;
somePair.second = "hello, world";
cout << somePair.first << " " << somePair.second << endl;
```

6.2 typename vs class in template specifications

For reasons well beyond the scope of this essay, you can write this:

```
template<class T>
int f(T a, int b) {
   // do something with a and b
}
```

which is *exactly* equivalent to this

```
template<typename T>
int f(T a, int b) {
   // do something with a and b
}
```

[1]DON'T WRITE THIS CLASS! The C++ standard library already comes with a template class called pair and you are wasting everyone's time by not using it!

from the compiler's perspective. It makes no difference. Some people prefer to write `class`, some like `typename`, some like to use one for some things and the other the rest of the time. My only advice is this: **pick a convention and stick with it.**

6.3 Template specializations

Say you have some template function like this:

```
template<typename T>
T f(T a, T b) {
   return a + b;
}
```

It works for 95% of your cases. It will work just fine for any class `T` which defines `operator+` appropriately. But imagine that the other 5% of your classes don't or can't define `operator+` for some reason. Maybe that operator is used for something else. Maybe it is a legacy class and you are not allowed to change it. Whatever. You don't have `operator+` for class `X`. *You can still make your template work for* `X`. You just need to specialize it!

Imagine that to add two instances of class `X` you must do this:

```
X a;
X b;
// initialize a and b
X c = a.sum(b); // NOT X c = a + b;
```

In that case we can specialize our `f` template like so:

```
// Generic case:
template<typename T>
T f(T a, T b) {
   return a + b;
}
// Special case:
template<>
X f<X>(X a, X b) {
   return a.sum(b);
}
```

Now when you call `f` on objects of type `X` the specialized version will get called. Spiffy.

6.4 Default template parameters

Fun fact about `C++`: functions can take default argument values. I don't know why, but this just isn't typically mentioned in your average intro programming courses. For example, this is legal `C++`:

```
int f(int a, double b = 1.2) {
   return (a + 3) * b;
}

int main(int argc, char** argv) {
   cout << f(3) << endl; // let b take default value of 1.2
   cout << f(3, -2.2) << endl; // override default value and make b = -2.2
   return 0;
}
```

Predictably, there are a few rules. The big one is this: if you make argument n take a default value, then argument $n+1$, $n+2$, $\ldots N$ must all also take default arguments. **THINK ABOUT WHY THAT MUST BE!** For example, this is not legal:

```
// NOT ALLOWED BECAUSE b TAKES NO DEFAULT VALUE WHILE a DOES
int f(int a = 2, double b, float c = 3.2) {
   return (a + 3) * b / c;
}
```

Along a similar line, templates can take default values. Think back to the `Matrix` template from before. If you are using `Matrix<double>` 95% of the time, then it makes sense to write this:

```
template<typename T = double>
class Matrix {

   private:

      T** data;

   public:

      Matrix() {
         // default constructor things
      }

      ~Matrix() {
         // destructor things
      }

      Matrix<T> add(Matrix<T> X) {
         // code which adds *this to x
      }
      // ...
};
```

With that default argument specified, we can now write

```
Matrix a; // No type param specified, so we default to double
Matrix<long double> b; // Type specified as long double
Matrix<complex> c; // Type specified as complex
```

Neato.

6.5 Template parameters need not be types

It is true. You can write this:

```
template<int N>
int f() {
   return N + 1;
}
```

Are you confused? You likely should be. What is the difference between that template function, and this function:

```
int g(int N) {
   return N + 1;
}
```

The answer is simple: **ALL TEMPLATE VALUES ARE COMPUTED AT COMPILE TIME!** The following code is illegal because it makes no sense:

```
int a;
cin >> a;
cout << f<a>() << endl;
```

The compiler has no idea which function to generate; the user would be deciding which function to compile *after* the compiler generated the code. You computer is powerful, but it is not a damn TARDIS.

On the other hand, these kind of template parameters give us the power to make certain decisions and computations *at compile time*, which may improve runtime speed. Say we have this code:

```
template<int N, int M>
int compile_time_max() {
   if (N < M) {
      return M;
   } else {
      return N;
   }
}
```

We could then write:

```
int main(int argc, char** argv) {
   cout << compile_time_max<1,4>() << endl;
   return 0;
}
```

Obviously this is a trivial case, but more complex cases are also clearly possible. Look at this for a minute:

```
template<unsigned int N>
unsigned int fact() {
   return N * fact<N-1>();
}
template<>
unsigned int fact<0>() {
   return 1;
}
```

What did I just manage to do?

7 Conclusion

So that is the basics of using templates. Now consider for a moment what you could do with new power. Remember building a dynamic array of `int`s? How about the linked list? We could easily generalize those codes to support arbitrary data types now. **BUT DON'T BOTHER!** The `C++` Standard Library already provides essentially all of the containers we have learned in this class, and then some. But that is a subject for another essay.

So that is it. We now know all about templates. **Hint**: if you ever find yourself thinking anything like "I know everything about templates," pick up *Modern C++ Design: Generic Programming and Design Patterns Applied* by Andrei Alexandrescu. Then cry.