An introduction to C++ template programming

Hayo Thielecke
University of Birmingham
www.cs.bham.ac.uk/~hxt

March 17, 2014

Abstract

These notes present an introduction to template programming in C++. They are aimed particularly at readers who are reasonably familiar with functional languages, such as OCAML. We will concentrate on features and idioms that are well understood from functional languages: parametric polymorphism and pattern-matching on function arguments.

Contents

1 Introduction

These notes are intended for readers who are familiar with basic procedural programming in a C-like syntax (such as Java) as well as functional programming (e.g., in OCAML). They were written with the students on my second-year C/C++ course in mind, who have been taught both Java and OCAML in their first year. See
`http://www.cs.bham.ac.uk/~hxt/2013/c-programming-language/`
However, readers may safely skip everything about OCAML when they do not find the comparison helpful.

In these notes, I am trying to go deep (to the extent possible in an undergraduate first course in C/C++) rather than broad. If there is something I find complicated or not absolutely necessary for the development at hand, I will try to be silent about it rather than muddy the waters. As the old saying goes, "somewhere inside C++ there is a smaller, cleaner language trying to get out", and until that language appears, all we can do is form subsets to our taste.

Templates are perhaps the part of C++ where compilers are most likely to deviate from the standard [C++], hence one may sometimes get different behaviours from different compilers. The examples below were tested with Xcode 5.0.2.

2 Templates in the design of modern C++

In object-oriented programming, the word "polymorphism" is often used for *dynamic dispatch* of methods calls (in C++ terminology: virtual functions). In Java, you get this dynamic dispatch behaviour for all overidden methods. See also `http://www.cs.bham.ac.uk/~hxt/2013/c-programming-language/objects-in-c.pdf`

The "polymorphism" in dynamic dispatch is completely different from the parametric polymorphism provided by templates. For comparison:

	Templates	Dynamic dispatch
When	compile-time	run-time
Typing	Type parameters	Subtyping
Efficiency	+ no runtime overhead - potential code bloat	- indirection via pointers at runtime
Related to	OCAML and Haskell polymorphism Java generics ML functors	Objective C messages Java methods

As C++ provides both templates and dynamic dispatch, they can be combined, which can become quite complex. On the other hand, it is an interesting question whether the increasing power of templates in C++ makes inheritance less important than it was claimed to be in the 1990s. The C++ standard library in now called the Standard Template Library (STL), and templates seem more central to its design then elaborate deep class hierarchies.

C++ can be thought of as composed of two layers of language constructs. The lower layer is a simple procedural language aimed at low-level data structures built mainly from `struct`s and pointers. That language is the “C” layer in “C++”. On top of it, the “++” layer, so to speak, provides abstraction mechanisms aimed at constructing complex software in a structured and type-safe way. The best known features of this high-level language layer that C++ puts on top of C are perhaps objects and classes, so that C++ is sometimes regarded, inaccurately, as an “object-oriented language”. While C++ historically evolved [Str94] from “C with classes”, the latest standard, C++11, defines a much more general language. Object-oriented programming is one of many programming styles supported by C++11. Note that using classes in C++ does not by itself constitute object-oriented programming. The term “class” is used rather widely for user-defined types in C++ and more or less interchangeably with `struct`. If one does not use inheritance and in particular virtual functions, there is nothing particularly object-oriented about such classes or structures. For instance, we may use a class with only static member functions as the best approximation that C++ provides to (tuples of) first-class functions, and structures may be plain old data tuples.

In these notes, we will concentrate on a subset of C++11 that may be seen as “C with templates”. Templates are by far the most advanced part of C++ and, perhaps surprisingly, the part of C++ that is closest to functional programming and lambda calculus. Templates form a higher-order, typed, purely functional language that is evaluated at compile time [Str12b, Str12a]. Note, however, that we will not pretend that C++ is, or ought to be, a functional programming language. The lower language level (inside functions and structures) can still be typical and idiomatic C, with assignments, pointers and all the rest; it is only the higher level of abstraction mechanisms that resembles functional programming.

Some introductions to templates put a lot of emphasis on their ability to perform arbitrary computations at compile-time. For instance, you can write a template that computes the factorial function *during* C++ compilation, and it might even output the result in compiler error messages for extra strangeness. However, in the latest C++11 standard, `constexpr` functions already provide compile-time functional computation. Here we will put greater emphasis to the relation of templates to type parametrization than their compile-time computation aspect, sometimes called meta-programming.

3 Templates and type parameters

The basic idea of C++ templates is simple: we can make code depend on parameters, so that it can be used in different situations by instantiating the parameters as needed. In C, as in practically all programming languages, the most basic form of code that takes a parameter is a function. For example, consider this C function:

```
int square(int n)
{
    return n * n;
}
```

Here the expression `n * n` has been made parametric in `n`. Hence we can apply it to some integer, say

```
square(42)
```

Templates take the idea of parameterizing code much further. In particular, the parameters may be types. For example, if `F` is a C++ template, it could be instantiated with `int` as the type, as in

```
F<int>
```

A typical example is the standard library template `vector`. By instantiating it with `int`, as in `vecor<int>`, we get a vector of integers.

Note the fundamental difference to the function `square` above. In the function, `int` is the *type of the argument*, whereas in `F<int>`, the argument is `int` itself.

There are two kinds of templates in C++:

1. Class templates
2. Function templates

These correspond roughly to polymorphic data types and polymorphic functions in functional languages.

To a first approximation, you can think of template instantiation as substitution of the formal parameters by the actual arguments. Suppose we have a template

```
template<typename T>
struct s {
 ... T ... T ...
 };
```

Then instantiating the template with argument `A` replaces `T` with `A` in the template body. That is, `s<A>` works much as if we had written a definition with the arguments filled in:

```
struct sA {
 ... A ... A ...
 };
```

The reality is more complex, but details may depend on the C++ implementation. A naive compiler may cause code bloat by creating and then compiling lots of template instantiations `s<A1>`, `s<A2>`, `s<A3>`, ... It is an interesting question how an optimizing compiler and/or a careful template programmer may avoid this risk of potential code bloat. On the other hand, because the argument replacement happens at compile time, there is no more overhead at runtime. Templates can produce very efficient code, in keeping with the aim of C++ to provide "zero-overhead" abstractions [Str12b].

In C, a similar form of replacement of parameters could be attempted using the macro processor. Templates, however, are far more structured than macros, which should be avoided in C++.

Readers who know λ-calculus may notice the similarity of template instantiation to β-reduction via substitution: we may read `template<typename T>` as analogous to λ `T`.

4 Parametric polymorphism

If you are familiar with a typed functional language such as OCAML [Ler13] or Haskell, you have already seen parametric polymorphism. That will make C++ templates much easier to understand.

The type of lists is polymorphic. There is a type of integer lists, a type of string lists, and so on. For example, here the OCAML compiler automatically infers that `[1; 2; 3]` is a list of integers:

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
```

Analogously, for a list of strings, we get:

```
# [ "foo"; "bar"; "qux" ];;
- : string list = ["foo"; "bar"; "qux"]
```

These examples are quite similar to `vector<int>` and `vector<string>` in C++. Note, however, that in C++ we often have to give the type parameters (`int` and `string`) explicitly rather than have the compiler infer them automatically.

When we define new types in OCAML, they may depend on a type parameter. For example, here is a definition of binary tree trees where the leaves carry data of type `'a`.

```
type 'a bt = Leaf of 'a
           | Internal of 'a bt * 'a bt;;
```

Such parametric type definitions correspond to template classes in C++.

Not only types but also function can be polymorphic. A standard example is the function `twice`:

```
# let twice f x = f(f x);;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
```

Polymorphic datatypes and functions can be combined. For example, we have both a polymorphic list type and functions like list reversal that operate on them:

```
val rev : 'a list -> 'a list
```

As we will see, a polymorphic data type corresponds to a class template in C++, which may be of the following form:

```
template<typename T>
struct S
{
    // members here may depend on type parameter T
    T data;          // for example a data member
    void f(T);       // or a member function
    using t = T;     // or making t an alias for T
};
```

Similarly, a polymorphic definition of a function `f` may be of the following form:

Click here to download full PDF material