An introduction to C++ template programming

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

January 1, 2016

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/2014/c-plus-plus/index.shtml`
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/2014/c-plus-plus/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:

```
template<typename T>
A f(B) // return type A and parameter type B may depend on T
{
    // function body may depend on T:
    T x;            // for example, variable x of type T
    T::g();         // or calling function g from class T
}
```

Note that the keyword `typename` in the parameter list of the templates gives the "type" of the parameter `T`. That is, `T` is itself a type.

There is a great deal more to templates than what has been sketched here, but polymorphic data structures and functions already provide powerful programming idioms familiar from functional programming. As usual in C++, there is a certain amount of syntactic complication, but there are also uses of templates that go far beyond what one can with polymorphism in OCAML.

5 Writing templates by abstracting over types

A good way to write template is to start with data structures or functions that work for a particular type and then generalizing that type to a template parameter.

Here is a type of linked lists of integers, together with a function that sums all the elements of such lists:

```
struct LinkedInt
{
    int head;
    struct LinkedInt* tail;
};
```

Listing 1: Linked list

```
int sumInt(struct LinkedInt *p)
{
    int acc = 0;
    while (p) {
        acc += p->head;
        p = p->tail;
    }
    return acc;
}
```

It is easy to see that a linked list works the same for all types. Hence we can replace `int` in listing 1 with `T` and make it a template parameter:

```
template<typename T>
struct Linked
{
    T head;
    Linked<T>* tail;
};
```

Note that `T` is a bound variable whose scope is the `struct`. We could have chosen any other name for it, e.g.

```
template<typename TypeParameter>
struct Linked
{
    TypeParameter head;
    Linked<TypeParameter>* tail;
};
```

6 Classes and pointers as template arguments

Generalizing the function is more involved. We do not just generalize `int` to `T` but we also need to generalize the operation `+` and the constant `0` in some way. There are several ways of doing so. The most general and arguably cleanest is to pass a class as a parameter, so that the class provides the required operations as static member functions.

The function template is:

```
template<typename T, typename Ops>
T fold(Linked<T> *p)
{
    T acc = Ops::initial();
    while (p) {
        acc = Ops::bin(acc, p->head);
        p = p->tail;
    }
    return acc;
}
```

Listing 2: Function template

For the case of integers, a class that implements the required operations can be defined as follows:

```
struct IntOps {
    static int initial() { return 0; };
    static int bin(int x, int y) { return x + y; }
};
```

Here is a main function for testing the function template on integers:

```
int main(int argc, const char * argv[])
{
    auto sumup = fold<int, IntOps>;

    Linked<int> x {3, nullptr };
    Linked<int> y {2, &x };

    std::cout << sumup(&y);

    return 0;
}
```

Now suppose we want to instantiate the linked list type and the function template to strings rather than integers. A class that implements string operations can be written as follows:

```
class StrOps {
public:
    static std::string initial() { return ""; };

    static std::string bin (std::string x, std::string y)
    {
        return x + y;
    }
};
```

Here is a main function for testing the function template on strings:

```
int main(int argc, const char * argv[])
{
    Linked<std::string> b = { "bar", nullptr };
    Linked<std::string> a = { "foo ", &b };

    auto sumupstr = fold<std::string, StrOps>;

    std::cout << sumupstr(&a) << "\n";

    return 0;
}
```

Exercise 6.1 In the function template in Listing 2, the template takes two parameters, a type T and operations on that type, which are given by the second parameter. Rewrite the code so that only one parameter is passed. Hint: C++ structs can contain types as members, so you can merge the first parameter into the second.

Exercise 6.2 In the OCAML standard library, there is:

```
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

The intended meaning is

```
fold_left f a [b1; ...; bn] = f (... (f (f a b1) b2) ...) bn
```

Generalize the function template so that it takes two type parameters `T1` and `T2`.

7 Function pointers as template arguments

Another possibility for passing the operations as template parameters to the function is to use value, rather than type parameters, in addition to the type parameter `T`. In particular, C function pointers can be passed as values.

```
template<typename T, T init, T (*bin)(T, T)>
T fold2(Linked<T> *p)
{
    T acc = init;
    while (p != nullptr) {
        acc = bin(acc, p->data);
        p = p->next;
    }
    return acc;
}
```

Listing 3: Function pointer as template argument

Here is how we can use the function template:

```
inline int sum(int x, int y)
{
    return x + y;

}

auto sumup2 = fold2<int, 0, sum>;
```

Although a function pointer as a template parameter looks like an indirect call, "it would be reasonable for the call to be inline" [VJ02, Section 22.5.5] given a sufficiently optimizing C++ compiler.

Another example is the old functional programming favourite `twice`. In OCAML, we write it as follows:

```
# let twice f x = f(f x);;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
# let inc n = n + 1;;
val inc : int -> int = <fun>
# twice inc 0;;
- : int = 2
```

With templates, we can translate the above as follows:

```
template <typename T, T (*f)(T)>
T twice(T x)
{
    return f(f(x));
}

int inc(int x)
{
    return x + 1;
}

int main(int argc, char * argv[])
{
    std::cout << twice<int, inc>(0);
}
```

Exercise 7.1 Explain where and how the OCAML type

```
('a -> 'a) -> 'a -> 'a
```

of `twice` is represented in the function template.

8 Member functions of template classes

A polymorphic data structure and a function operating on it can be defined separately, as is typical in functional programming. In a more object-oriented style, data and operations on it can be packaged together in a class, by defining member functions of the class.

Now we can use the function object together with a function template:

```
template<typename T>
struct Linked
{
    T data;
    Linked<T>* next;
    void apply_each(void(*f)(T));
};

template<typename T>
void Linked<T>::apply_each(void(*f)(T))
{
    T* p = data;
    while(p) {
        f(p->data);
        p = p->next;
    }
}
```

Note that the member function is defined outside the class declaration. Hence it is outside the scope of the template parameter `T` of `struct Linked`. As the function definition needs to be in the scope of the template parameter, it has to be preceded by `template<typename T>` to avoid the variable `T` being unbound.

Exercise 8.1 Does the template parameter `T` in the member function definition have to be the same as in the class declaration, or can it be renamed, say to `T2`?

9 Function objects and templates

A function object is an object that can be used like a function. One of its member functions overloads the function call syntax `()`. Such an object can have its own data members. We can create function object dynamically, unlike C functions. In functional programming terms, function objects simulate *closures*. Function objects are often used with templates. Objects *cannot* be template parameters, only function parameters. Roughly speaking, this restriction make some sense, because objects exist at run-time and template are processed entirely at compile-time.

Here is an example:

```
// class for curried addition function
class cadd {
private:
    int n;
public:
    cadd(int n) { this->n = n; }

    int operator() (int m) { return n + m; }
};

int main(int argc, const char * argv[])
{
    // make a new function object in local var
    cadd addfive(5);

    cout << addfive(7);
}
```

This prints 12.

Function objects can be parameters to functions. Here is an example where the template type parameters can be inferred automatically by the compiler:

```
template <typename T, typename Op>
T twice(T x, Op f)
{
    return f(f(x));
}

int main(int argc, const char * argv[])
{
    cadd addfive(5); // create function object

    cout << twice<int, cadd>(10, addfive)
         << endl;
         // <...> can be omitted
    cout << twice(10, addfive)
         << endl;
}
```

This prints 20 twice.

C++11 adds a more concise syntax for function objects: so-called "lambda expression" [Str12a, Section 11.4]. The word "lambda" comes from the λ-calculus.

Here is an example:

```
    [=] (int x) { return x + a; }
```

It is like

```
    fun x -> x + a
```

in OCAML or

$$\lambda x.x + a$$

in λ-calculus. The syntax `[=]` indicates that all variables in scope should be "captured" and become part of the closure. The variable `a` needs to be in scope.

We can use a "lambda expression" as a parameter, as follows:

```
    int foo = 10;

    cout <<
    twice(1, [=] (int x) { return x + foo; })
    << endl;;
```

This prints 21.

C++ tries to approximate functional programming, more or less analogous to the following:

```
# let cadd n m = n + m;;
val cadd : int -> int -> int = <fun>
# cadd 5 7;;
- : int = 12

# List.map (cadd 5) [1; 2; 3];;
- : int list = [6; 7; 8]
```

Compare iterators in the C++ Standard Template Library.

10 Template `std::function`

The template `std::function` gives general function types. They are useful for typing lambda expressions. For example:

```
#include <iostream>
using namespace std;

function<int(int)> f(int n)
{
    return [=] (int m) { return n + m; };
}

int main(int argc, const char * argv[])
{

    auto g = f(2);
    cout << g(3) << endl ;
    cout << g(4) << endl ;
}
```

This prints 5 and 6.

11 Void pointers and polymorphism

To some extent, the flexibility of templates can be approximated in C by accessing data via pointers. It is instructive to compare the use of template type parameters to the (much more unsafe) use of void pointers in C.

Here is a linked list type that stores a pointer to the data:

```
struct Linked
{
    void* data;
    struct Linked* next;
};
```

Here is a function that operates on such lists:

```
void* fold(struct Linked *p, void *initial,
           void *(*bin)(void *x, void *y))
{
    void *acc;
    acc = initial;
    while (p) {
        acc = bin(acc, p->data);
        p = p->next;
    }
    return acc;
}
```

In addition to the linked list, the function takes a function pointer as a parameter.

Here is a function that can be used when we want to instantiate to integers:

```
void *ptr2sum(void *x, void *y)
{
    int *p;
    p = malloc(sizeof(int));
    *p = *(int *)x + *(int *)y;
    return p;
}
```

Here is a main function for testing the code above:

```
int main(int argc, const char * argv[])
{
    int zero = 0;
    int one = 1;
    int two = 2;
    int three = 3;

    struct Linked x = {&three, NULL };
    struct Linked y = {&two, &x };
    struct Linked z = {&one, &y };

    int *r = fold(&z, &zero, ptr2sum);
    printf("%d\n\n", *r);
    return 0;
}
```

The main difference between templates and the old style of void pointer polymorphism in C is that the type system cannot ensure that void pointers are used consistently. It is entirely up to the programmer to make sure that a void pointer points at data of the expected type when it is dereferenced.

Exercise 11.1 Suppose you have a linked list of pointers to floats and you try to sum it up by using a function that works on pointers to integers. What will happen? Is it reasonably to expect the compiler to detect the problem; or that type conversions are done automatically?

12 Template template parameters

There is still more to templates that should be easy for functional programmers. Templates can themselves be arguments to templates.

Templates as template arguments may arise quite naturally when we pass a container class like vector to a template.

```
template<template<typename>class Cont>
struct UseContainer {
    Cont<int> ic;
    // now we can put int data into ic
};
...
UseContainer<vector> uc;
```

Passing the container and letting the template apply it as it wishes can be a more flexible design than passing the container already instantiated to `int` [VJ02]. In the Standard Template Library, the template class `vector` takes an allocator as its second parameter, which is itself a template parametric in a type [Str12a, Section 31].

Of course, in functional programming it is straightforward for a function to take another function as a parameter, for instance we can write:

```
# fun f -> f 42;;
- : (int -> 'a) -> 'a = <fun>
```

13 Type-safety and templates

One of the killer apps, so to speak, of templates are *type-safe* containers. Suppose we want to store objects of different types in some container, such as an array or a list. In Java, one could try to use an array of `Object` as a universal container that can hold objects of any class type. Due to subtyping, you can indeed store objects of any class type `A` in such an array. The problem is what you get back when reading from the array: the type is `Object`, not `A`. To use the object, for example calling one of is methods, one may have to cast it to `A`. The need for casting shows that the array of objects is not type-safe. In fact, it has similar problems to the use of void pointers in C. The type-safe container problem was solved when generics were added to Java. With generics or C++ templates, a container such as `vector<A>` is specific to the type `A`. That means you are guaranteed to get back an `A`, without the need for unsafe casting. More generally, this situation illustrates the difference between *subtyping*, as found in typed object-oriented languages, and *parametric polymorphism*, as found in functional languages and also C++ templates.

One of the advantages of templates is that they allow compile-time type checking, unlike void pointers as a kind of poor man's polymorphism for example. However, templates do not quite provide the same degree of type safety as polymorphism in OCAML. Concerning template type checking, Stroustrup [Str12b] writes that

> [...] the arguments to a template are unconstrained and only their instantiations are type checked. This [...] leads to late (link-time) type checking and appallingly poor error messages.

We could define a template that formalizes the interface for the parameter of the template in Listing 2:

```
template<typename T>
struct AbsOps {
    static T initial();
    static T bin(T x, T y);
};
```

In Java, interfaces are used in this way to type parameters in generics. This is sometimes called *bounded* polymorphism. In contrast, in C++ the arguments of templates need not be derived from any base class. The C++ type system is not expressive enough to capture what is and is not a valid argument for a given template. There may be type errors that are only caught when a template is applied, not when it is declared (although this is still at compile time and not at run time). The design of generics in Java had more input from type theory than the evolution of C++ templates. Moreover, C++ templates are a more general mechanism than Java generics, so it is to be expected that typing them is harder. (Some current research involves so-called concepts and traits.)

Passing a function pointer as a template parameter can give more type safety than passing a class. For example, the polymorphic function `twice` can be written as:

```
template <typename T, T (*f)(T)>
T twice(T x)
{
    return f(f(x));
}
```

Here the typing of the template parameters, and `f` in particular, ensures that the body of the function,

```
f(f(x))
```

is well typed whenever the template arguments match the types in the parameter list. By contrast, if we pass the function `f` as part of a class, the template parameter list looks like this:

```
template <typename T, typename F>
```

and we would have to look inside the body of the function to see if operations from `F` are applied correctly.

There is at least one case where the typing of C++ templates is *less* powerful than Java generics: template member functions cannot be virtual, whereas

in Java all functions are virtual, including parametrically polymorphic ones. That means that a cleaner typing of the Visitor pattern can be programmed in Java [BT06] than is possible in C++.

The typing of higher-order templates has some aspects of simply-typed λ-calculus, so that it catches some type errors. Specifically, errors are detected that are due to parameters and arguments not matching at the right level of templates, template templates, template template templates, etc. The standard example of something that cannot be typed[1] is a self-application, $f(f)$. Here f has some type $A \to B$, so that it has to be applied to something of type A. However, it is applied to f, which has type $A \to B$ and not A as required. In OCAML, we get this message if we try to type a self-application:

```
# fun f -> f f;;
Error: This expression has type 'a -> 'b
       but an expression was expected of type 'a
       The type variable 'a occurs inside 'a -> 'b
```

The same issue arises if we attempt to apply a template to itself.

```
template <typename T>
struct f {
};

f<f> s; // error: template argument required
```

To fix the problem, we could try to make the parameter a template template:

```
template <template<typename>class T>
class f2 {
};
```

Now `f2<f>` is well typed, but not the self-application `f2<f2>`. The latter produces a type error at the level of template templates. In sum, this part of the template type system is fairly clean and close to functional programming.

14 Template typing versus derived class typing

It is instructive to compare and contrast the typing of templates with that of C++ virtual functions using dynamic dispatch. (In Java, all method overriding behaves like C++ virtual functions.) Virtual functions are sometimes called *runtime* polymorphims, and they are *statically typed* using a form of subtyping (derived classes have a subtype of their base class).

Here is a standard example of using virtual functions.:

```
class animal // base class
{
public:
    virtual void speak() { printf("Grunt."); }
    // default if not overriden in derived class
};
```

[1] at least not in the simply-typed λ-calculus

```
class baboon : public animal
// subtyping: baboon can be used where animal is expected
{
public:
    void speak() { printf("I R Baboon!"); }
    // override virtual function
};

class weasel : public animal
{
public:
    void speak() { printf("I M Weasel!"); }
};

int main(int argc, const char * argv[])
{
    animal *p;
    if (random_bool())    // cannot predict at compile time
        p = new baboon(); // OK due to subtyping
    else
        p = new weasel(); // OK due to subtyping
    p->speak(); // baboon or weasel? Either works
}
```

Listing 4: Overriding virtual function `speak`

Suppose we have a pointer `p` of type `animal*`. The pointer may in fact point to an object of the derived class `baboon`. When we call `p->speak()`, the appropriate function for `baboon` is selected at run time (the *dynamic* in dynamic dispatch). All concrete classes derived from `animal` must implement a member function `speak`. In that sense, the C++ type system ensures that nothing goes horribly wrong because we try to run some code that does not exist. (But using casts you can do so, and get exciting errors at runtime, including the dreaded arbitrary code execution.)

Now compare the above to the situation with templates. Suppose we have a function template that requires its arguments to provide a `speak` function:

```
template<typename T>
void let_speak()
{
    T::speak();
}
```

We can instantiate the template using a class that has a `speak` function:

```
class baboon
{
public:
    static void speak() { printf("I R Baboon!"); }
};
```

Note that here we need a `static` function. Then the following works:

```
let_speak<baboon>();
```

But now suppose we have some class completely unconnected to `baboon`:

```
class academic
{
public:
     static void speak() { printf("lambda lambda lambda"); }
};
```

As there is no base class in this case, there is no potential for subtyping and dynamic dispatch. Again, instantiating the template works fine:

```
let_speak<academic>();
```

There is no requirement for template arguments to have a common base class, very much in contrast to Listing 4. A problem only arises when we provide a template argument that does not provide the required function at all:

```
class plant { void vegetate(); };
```

Then we get a compiler error when attempting to instantiate the template with it:

```
let_speak<plant>(); // error at template instantiation
```

The typing of template arguments and derived classes is an instance of an old question in programming language design: definitional versus structural type equivalence. Suppose we have two different definitions of types, but they contain the same information (like `speak` above). Should the types be regarded as interchangeable? Templates and classes take a different view: the former say yes, while the latter only when the types are related via the class hierarchy (subtyping).

Long before C++ and objects, C already chose the definitional view in its type system:

```
struct cm { float val; };

struct inch { float val; }; // same or different?

int main(int argc, char * argv[])
{
     struct cm x;
     struct inch y;
     x = y;                    // error: incompatible types
     x.val = 2.54 * y.val;     // OK, same type float
}
```

15 Specialization as pattern-matching on template arguments

In OCAML, the canonical way to process tree-shaped data structures is to use pattern matching on the constructors. For example, suppose we have a binary tree type defined as follows:

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

Then a function can pattern-match on the cases of `Leaf` and `Internal`. In each case, the pattern also matches variables. On a pattern `Leaf x`, the compiler knows that `Leaf` is the constructor and `x` is a variable bound in the pattern. Similarly, in a pattern `Internal (left, right)`, the two variables `left` and `right` are bound, and we may call the function recursively on them to process the tree.

The pattern-matching in C++ templates works similarly to that in functional languages. However, instead of user-defined data types, the language provides a special built-in tree-shaped data type that can be matched: the language of type expressions. For instance, the C type of arrays of pointers to integers, `int *[]`, corresponds to a tree where the type constructor `[]` is at the top, followed by the type constructor `*` for pointers and finally the type `int` at the bottom. The C++ compiler knows these type constructors, as they are part of the language.

Exercise 15.1 Draw the tree for the C type expression `float(*[])(int*)`.

The motivation for template specialization is to provide better implementations for particular types. A container class cloud be specialed for type `bool` to chose a more efficient bit vector representation [VJ02]. In the Thrust template library [HB11] for GPU programming, the template for sorting is specialized in the case of `int` to chose a radix sort implementation that is more efficient than the general purpose sorting algorithm

If there is an implementation that works for all pointer types, one can write a specialization that handles all types of the form `T *`:

```
template<typename T>
struct C<T*>
... // implementation for pointer types
```

How does the compiler determine what in a pattern is a variable and what is not? In some languages, constructors and variables are distinguished lexically. In C++, the variables in a pattern are introduced beforehand. Somewhat confusingly, this variable introduction is done by writing `template` again.

As an example, we will solve Kernighan and Richie's [KR88] exercise of translating C type expressions into English in a labour-saving way without having to write a parser. Template specialization matches the type expression and recursively constructs a string that describes the type in English, e.g. `int*` should produce "pointer to int".

```
template<typename T>
struct NameofType;

template<typename T, typename S>
struct NameofType<T(*)(S)> {
    static void p()
    {
        std::cout << "pointer to function returning a ";
        NameofType<T>::p();
        std::cout << " and taking an argument of type ";
        NameofType<S>::p();
    }
};
```

```
template<>
struct NameofType<int> {
    static void p()
    {
        std::cout << "int";
    }
};

template<>
struct NameofType<float> {
    static void p()
    {
        std::cout << "float";
    }
};

template<typename T>
struct NameofType<T*> {
    static void p()
    {
        std::cout << "pointer to ";
        NameofType<T>::p();
    }
};

template<typename T>
struct NameofType<T[]> {
    static void p()
    {
        std::cout << "array of ";
        NameofType<T>::p();
    }
};
```

For example, we can instantiate the template to some complicated type expression like `float(*[])(int*)` and then call its `p` function:

```
NameofType<float(*[])(int*)>::p();
```

The function call prints the following:

```
array of pointer to function returning a float
and taking an argument of type pointer to int
```

The parsing of the type expression is done by the C++ compiler at compile time. Some cases, e.g. `long` and functions of more than one argument, are left as exercises for the reader.

For a more substantial use of templates that pattern-match on types, see [VJ02, Section 19].

16 Dependent types via templates

In parametric polymorphism, a type can depend on another type. In *dependent types*, a type can depend on a value. For example, there could be a type of

vectors with exactly n elements, where n is an integer. Such dependent types cannot be defined in core OCAML, but they exist in some research languages, for example Agda [Nor09].

With templates, we can define a type of square $n \times n$ matrices, depending on n.

```
template<typename T, int n>
struct squareMatrix {
    T a[n][n];
};

squareMatrix<float, 2> sqmat =
{ { { 1.0, 2.0 },
    { 3.0, 4.0} }
};
```

In a more advanced use of templates for matrices, we let the *dimension* of the matrix depend on a parameter n. This (very rough) example uses recursive template specialization.

```
template<typename T, int n>
struct ndimMatrix;

template<typename T>
struct ndimMatrix<T, 0>
{
    T m[];
};

template<typename T, int n>
struct ndimMatrix
{
    ndimMatrix<T,n - 1> m[];
};
```

For a fully developed design of n-dimensional matrices using templates, see Chapter 29 of Stroustrup's C++ book [Str12a]. Recursion over a dimension parameter [VJ02, Section 17.7] can also be used for loop unrolling.

17 Templates and inheritance

In C++, templates can be combined with inheritance in various ways, giving some subtle and surprising interactions.

It is possible to use a template parameter as a base class:

```
template<typename T>
struct D : public T {
    ...
};
```

Inheriting from a template parameter is sometimes called a "mixin" [SB01].

Templates can be used to set up a recursive form of inheritance, as follows:

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

class C : public F<C> { ... };
```

This idiom is known as the "curiously recurring template pattern" (CRTP) in the literature, although descriptions of it vary in details [VJ02, Section 16.3]; [Str12a, Section 27.4]. Vandevoorde and Josuttis [VJ02, Section 16.3] present three variations on the CRTP.

Here is an example of a tree structure [Str12a, Section 27.4.1] defined via templates and inheritance:

```
template<typename T>
struct F {
    T* left;
    T* right;
};

template<typename V>
struct N : F<N<V>>
{
  V v; // the payload data inside nodes
};
```

Here the template by itself does not construct a recursive data structure (tree), as the pointers are to the type of its parameter, not the structure itself. The recursion is only set up by the inheritance. (This is somewhat reminiscent of the way that recursion can be defined via self-application in the untyped lambda calculus.)

Whether the above indirect recursion example is convincing enough to justify its complication is perhaps debatable, as there is a much more straightforward way to build such trees using parametric polymorphism and no inheritance:

```
template<typename T>
struct F {
    F* left;
    F* right;
    T v; // the payload data inside nodes
};
```

18 Higher-order templates (for λ-calculus fans)

Templates are a pure functional language, albeit in a strange and forbidding syntax. Writing some template idioms as analogous lambda calculus terms makes it easier to see what is going on.

Templates can be passed as parameters to other templates. For instance:

```
template<template<typename>class f, typename x>
struct Twice {
    using result = f<f<x>>;
};
```

This example is analogous to the lambda term

$$\lambda(f, x).\, f(f\, x)$$

Templates can take several parameters in the same parameter list, just like a C function:

```
template<typename x, typename y>
struct Pair{
    x fst;
    y snd;
};
```

By analogy with functional programming, we might try to write a *curried* template that takes one parameter after the other. However, it is not syntactically correct to have multiple `template` constructs at the beginning of a definition:

```
template<typename x>  // syntax error
template<typename y>  // template keyword not allowed here
struct CurriedPairWrong{
    x fst;
    y snd;
};
```

Nonetheless, we can write "curried" templates, albeit with some notational complication. Templates can be the result of templates (wrapped into a `struct`).

```
template<typename x>
struct k1 {
  template<typename y>
    struct k2 {
        using type = x;
    };
};

k1<int>::k2<float>::type n = 42;
```

This example is analogous to the K combinator in lambda calculus:

$$\lambda x.\, \lambda y.\, x$$

Here is a more involved example of higher-order templates: currying.

```
template<template<typename, typename>class F>
struct curry {
    template<typename A>
    struct t1 {
        template<typename B>
        struct t2 {
            using t = F<A,B>;
        };
    };
};

template<typename A, typename B>
struct prod {
```

```
    A fst;
    B snd;
};
```

For example, if we apply `curry` to `prod`, we get a template, which we can then apply to `int` and then to `double`. The result is a type, which we can then use to declare a variable `q`.

```
curry<prod>::t1<int>::t2<double>::t q { 1, 2.0 } ;
```

Compare the above to a function that curries its argument in λ-calculus:

$$\lambda f.\lambda a.\lambda b.f(a,b)$$

Exercise 18.1 (If you know enough λ-calculus) Prove that there is a β-reduction

$$(\lambda f.\lambda a.\lambda b.f(a,b))\, p\, a\, b \to_\beta p(a,b)$$

The point of the preceding exercise should become clearer with the next one:

Exercise 18.2 Argue that

```
curry<prod>::t1<int>::t2<double>::t
```

should be equivalent to

```
struct prod {
    int fst;
    double snd;
};
```

You may assume that template instantiation works by argument substitution. For instance, `prod` is substituted for `F` in `curry`, and `A` and `B` are then substituted by `int` and `double`.

Note that the templates operate at the level of types, not values, so we have a higher-order λ-calculus like Fω. Convincing examples that use the full power of higher-order templates are not necessarily easy to find, so we content ourselves here with noting out that these powerful abstraction mechanisms are available in C++. The Boost template library [AG04] makes extensive use of templates for type computations, based on an encoding of untyped λ-calculus in templates.

References

[AG04] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison Wesley, 2004. 24

[BT06] Peter Buchlovsky and Hayo Thielecke. A type-theoretic reconstruction of the visitor pattern. In *21st Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)*, volume 155 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 309–329. Elsevier, 2006. 16

[C++] Working draft, standard for programming language C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`. 2

[HB11] Jared Hoberock and Nathan Bell. Thrust: a productivity-oriented library for CUDA. In *GPU Computing Gems*. Morgan Kaufman, 2011. 19

[KR88] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Prentice-Hall, 1988. 19

[Ler13] Xavier Leroy. The ocaml system, release 4.01. documentation and users manual. `http://caml.inria.fr/pub/docs/manual-ocaml/`, 2013. 5

[Nor09] Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009. 21

[SB01] Yannis Smaragdakis and Don Batory. Mixin-based programming in C++. In *Generative and component-based software engineering*, pages 164–178. Springer, 2001. 21

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison Wesley, 1994. 3

[Str12a] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 2012. 3, 11, 14, 21, 22

[Str12b] Bjarne Stroustrup. Foundations of C++. In *ESOP 2012*, pages 1–25. Springer, 2012. 3, 4, 15

[VJ02] David Vandevoorde and Nicolai M Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002. 9, 14, 19, 20, 21, 22