7 Templates and Generic Programming

Templates enable us to write generic, or parameterized, code. The basic idea is simple but some of the implications (at least for C++) are subtle.

You can find more information on the material in this section in (Weiss 2004, Chapter 7), (Koenig and Moo 2000, Chapter 8), or (Prata 2012, Chapter 14). For a more detailed and complete description of templates, (Vandevoorde, Josuttis, and Gregor 2017) is recommended.

7.1 Template Functions

Consider these three functions:[26] 216

note that you cannot write a swap function like this in Java, since it does not provide call-by-reference

```
void Swap(char & x, char & y)      // Function (1)
{
   char t = x; x = y; y = t;
}

void Swap(int & x, int & y)
{
   int t = x; x = y; y = t;
}

void Swap(double & x, double & y)
{
   double t = x; x = y; y = t;
}
```

These functions perform the same task for different types. Since C++ allows overloading, we could use all three in the same program, but there does appear to be some redundancy.

7.1.1 Type Parameters

We can use templates to avoid source code redundancy. We replace the previous three definitions by a ***template function declaration*** as shown in Figure 72 on the following page. 217

[26] The name `Swap`, rather than `swap`, was chosen to avoid confusion with the standard library function.

```
template <typename T>
void Swap(T & x, T & y)
{
    T t = x;
    x = y;
    y = t;
}
```

Figure 72: Changing `Swap` into a template function

The new function can be called in the same way as the previous versions. It is not necessary to specify the type of the arguments, because the compiler already has this information. The
218 code

```
char c1 = 'a';
char c2 = 'b';
Swap(c1, c2);
cout << c1 << ' ' << c2 << endl;

int m = 3;
int n = 5;
Swap(m, n);
cout << m << ' ' << n << endl;
```

prints

```
b a
5 3
```

219 To compile the call `Swap(c1, c2)`, the compiler must perform the following steps:

1. Infer the types of the arguments `c1` and `c2` (in this case: `char` and `char`).
2. Look for functions named `Swap`.
3. Find (in this case) a template function `Swap`.
4. Check that the substitution $T =$ `char` matches the call.
5. Generate source code for the function `Swap<char>` (which should be the same as Function (1) above).
6. Compile the generated source code.
7. Generate a call to this function.

The compiler detects errors in the template code, if there are any, at Step 6., when it compiles the code obtained by expanding the template. This means that errors in template code are reported ***only if the template is used***. You can write all kinds of rubbish in a template declaration and the compiler won't care if you never use the template.

The process of deriving an actual function from a template declaration is often called "instantiating" the template. This usage is confusing, because we also talk about objects obtained by instantiating a class. These notes use the expression ***applying a template*** or ***the application of a template***, by analogy with "applying a function".

Although templates remove redundancy in the source code, they do not affect the object code. If the program calls `Swap` with N different argument types, there will be N versions of `Swap` in the object code.[27]

The following template function `Max` returns the greater of its two arguments.[28] 220

```
template <typename T>
const T Max(const T & x, const T & y)
{
    return x > y ? x : y;
}
```

We would expect this function to work with various types, and indeed it does. Each of the following statements compile and execute correctly:

```
cout << Max(7, 3) << endl;
cout << Max('a', '5') << endl;
cout << Max(7.1, 3.3) << endl;
```

This statement ***appears*** to execute correctly:

```
cout << Max("COMP", "6441") << endl;
```

But further investigation reveals problems. For instance, executing 221

```
cout << Max("apple", "berry") << endl;
```

displays `berry` but executing 222

```
cout << Max("berry", "apple") << endl;
cout << Max("apple", "berry") << endl;
```

displays `apple` twice. Even worse, the statement 223

```
cout << Max("berry", "cherry") << endl;
```

gives a compiler error:

```
error C2782: 'T Max(const T &,const T &)' :
   template parameter 'T' is ambiguous
```

224 To find out what is going wrong, we add a line to `Max`:

```
template <typename T>
const T Max(const T & x, const T & y)
{
    cout << "Max called with " << typeid(x).name() << endl;
    return x > y ? x : y;
}
```

In order to compile this, we must add the directive

```
#include <typeinfo>
```

225 Executing

```
cout << Max("apple", "berry") << endl;
cout << Max("cherry", "orange") << endl;
```

displays

```
Max called with char const [6]
apple
Max called with char const [7]
orange
```

and reveals the problem: we cannot compare strings of different lengths because they have different types. Also, `Max` is not comparing the strings; it is comparing the addresses of the strings (i.e., the pointers).

There are no good, general solutions to problems like this. For `Max`, the best thing to do is to
226 define a non-template version for `string`s:

```
const string Max(const string & x, const string & y)
{
    return x > y ? x : y;
}
```

When the compiler encounters `Max("apple", "berry")`, it will consider the `string` version a better match than the template version.

7.1.2 Missing Functions

Suppose, however, that we define our own class `Widget` as follows: 227

```
class Widget
{
public:
    Widget(int w) : w(w) {}
private:
    int w;
};
```

Note that the private variable `w` is set using a *constructor initializer*, which we discussed in the previous lecture.

Attempting to use `Max` with widgets

```
Max(Widget(1), Widget(2));
```

gives several error messages, including this one:

```
error C2676: binary '>' : 'const Widget' does not define
    this operator or a conversion to a type acceptable
    to the predefined operator
```

It is easy to see what has happened: the compiler has generated the function 228

```
const Widget Max(const Widget & x, const Widget & y)
{
    return x > y ? x : y;
}
```

and has then discovered that `Widget` does not implement `operator>`. The correction is also straightforward: we just have to add the function

```
friend bool operator>(const Widget & left, const Widget & right)
{
    return left.w > right.w;
}
```

to the declaration of class `Widget`.

Ensure that template arguments satisfy the requirements of the corresponding template parameter.

[27] Note that this is quite different from Java's *Generics*, where only one object copy of the generic code is used for all type parameters (the generic types are removed at compile-time through a process called *type erasure).*

[28] The name `Max` is used to avoid confusion with the library function `max`.

7.1.3 Conversion Failure

229 The use of templates prevents some of the conversions that we expect. If we declare

```
const double Max(const double & x, const double & y)
{
    return x > y ? x : y;
}
```

then the following calls all compile and execute correctly:

```
Max(1.2, 3.4);
Max(1, 3.4);
Max(1, 3);
```

The first call works because the types match exactly, and the other two calls work because the compiler includes code to convert 1 and 3 from `int` to `double` before the function is called.

With the template version, however, the compiler does not allow the second call. It matches `Max(int,int)` and `Max(double,double)` to the template pattern, but `Max(int,double)` does not match and the compiler will not insert conversions to make it match. 230

231 There are several ways to make the second call work properly:

- We can cast the argument that is causing the problem:

  ```
  Max(static_cast<double>(1), 3.4);
  ```

- We can specify the template argument explicitly:

  ```
  Max<double>(1, 3.4);
  ```

- We can avoid calls of the form `Max(1, 3.4)` with mixed-type arguments.

This is dangerous in this example, since the result can now be different depending on whether you call `(int, double)` or `(double, int)`!

- We can define a template for mixed types using casting:

  ```
  template <typename T, typename U>
  const T Max(const T & x, const U & y)
  {
      return x > static_cast<T>(y) ? x : static_cast<T>(y);
  }
  ```

- We can provide ***both*** a template version and a specialized version of the function. C++ will always pick the function that provides the most closely matched argument types. 232

7.1.4 Non-type Parameters

Template parameters are not restricted to class types. We can also use integral types (`char`, `short`, `int`, `long`, ...) as parameters. This function has an integer template parameter.[29] 233

```
template<int MAX>
int randInt()
{
    return rand() % MAX;
}
```

and is used like this:

```
// Throwing dice
for (int i = 0; i != 20; ++i)
    cout << setw(2) << randInt<6>() + 1;
cout << endl;
```

The argument corresponding to a non-type template parameter can be a constant, but it cannot be a variable: 234

```
const int MAXRAND = 100;
.... randInt<MAXRAND>() ....    // OK

int MAXVAR = 100;
.... randInt<MAXVAR>() ....     // Compiler error
```

Of course, we could have written this function without using templates: 235

```
int randInt(const int MAX)
{
    return rand() % MAX;
}
```

Then we would call it in the usual way: `randInt(6)`.

The difference between the two versions of `randInt` is that the template version substitutes the integer ***at compile time*** whereas the conventional version substitutes the integer ***at run time***. In this case, the difference is slight – the template version probably runs slightly faster than the conventional version but the difference will hardly be noticeable – but may be significant in more realistic situations.

templates: design trade-off between compile-time and run-time overhead

7.2 Template Classes

Classes can be parameterized with templates; the notation is similar to that of function templates.
236 Here is a simple class for 2D coordinates, in which each coordinate consists of two `float`s:

```
class Coordinate
{
public:
    Coordinate(float x, float y) : x(x), y(y) {}
    void move(float dx, float dy);
private:
    float x;
    float y;
};
```

To parameterize this class, we:

1. Write `template<typename T>` in front of it;
2. replace each occurrence of `float` by `T`; and
3. – important! – within the declaration, replace occurrences of the class name C by C`<T>`.

Performing these steps for class `Coordinate` yields the following declaration. Note that, whenever the class name (`Coordinate`) appears ***within*** the declaration, it must have the argument `<T>`:

```
template<typename T>
class Coordinate
{
public:
    Coordinate<T> (T x, T y) : x(x), y(y) {}
    void move(T dx, T dy);
private:
    T x;
    T y;
};
```

Unlike functions, the compiler cannot infer the argument type for classes. Whenever we create an application of a template class, we must provide a suitable argument. The following statements
237 create ***two different classes*** and one instance of each:

```
Coordinate<int> p(1, 2);
Coordinate<float> q(3.4, 5);
```

The integer argument `5` is acceptable for the `Coordinate<float>` constructor because the template application is explicit: the compiler knows that `float` values are expected, and inserts the appropriate conversion.

Here is how `move` is defined for class `Coordinate`:

[29] This is a terrible way to generate random integers! We will consider better ways later.

```
template<typename T>
void Coordinate<T>::move(T dx, T dy)
{
    x += dx;
    y += dy;
}
```

In general, if the definition of a member function for a template class uses the template parameter, we must:

1. Write `template<typename T>` before the function; and
2. write *C*`<T>` wherever the class name *C* is needed.

Like functions, template classes can have non-type template parameters, provided that the parameters have integral types. Instances of the class must be provided with constant arguments of appropriate types.

7.2.1 A Template Class for Coordinates

238

Figure 73 on the next page shows part of a template class for coordinates. It is parameterized by `Type`, the type of a coordinate element, and `Dim`, the dimension of the coordinates. A typical declaration would be 239 240 241

```
Coordinate<double, 3> c;
```

If the program uses many coordinates of this type, we would probably define a special type for them, to reduce the amount of writing required and improve the clarity of the program:

```
typedef Coordinate<double, 3> coord;
```

Some points to note about the declaration of `Coordinate`:

- There is a default constructor that sets the elements of the coordinate to zero. The compiler's default constructor would leave the elements uninitialized.
- The function `operator[]` returns a reference to a coordinate element. This function allows a user to get or set any element. This function is called by code like this:

  ```
  x = c[0];
  c[1] = y + 1;
  ++c[2];
  ```

- It is good practice (as we will discuss later) to provide two versions of `operator[]`, one returning an Lvalue (as done here) and the other returning an Rvalue.
- The function `operator[]` performs a range check and aborts the program if the range check fails. It would probably be better to handle a range check error by throwing an exception (which we will also discuss later).

```
template<typename Type, int Dim>
class Coordinate
{
public:
Coordinate<Type, Dim>()
{
    for (int d = 0; d != Dim; ++d)
       c[d] = 0;
}

Type & operator[](int i)
{
    assert(0 <= i && i < Dim);
    return c[i];
}

friend Coordinate<Type, Dim> operator+(
      const Coordinate<Type, Dim> & left,
      const Coordinate<Type, Dim> & right )
{
    Coordinate<Type, Dim> result;
    for (int d = 0; d != Dim; ++d)
        result.c[d] = left.c[d] + right.c[d];
    return result;
}

friend ostream & operator<<(ostream & os,
                            const Coordinate<Type, Dim> & coord)
{
    os << '(';
    for (int d = 0; d != Dim; ++d)
    {
        os << coord.c[d];
        if (d < Dim - 1)
            os << ", ";
    }
    return os << ')';
}

private:
    Type c[Dim];
};
```

Figure 73: Part of a template class for coordinates

- Several of the functions have `for`-loops for the range `[0,Dim)`. A good compiler might optimize these away for small values of `Dim` (this optimization is a special case of ***loop unrolling***). If we were worried about the overhead of a `for`-loop, we could rewrite the code using `if` or `switch` statements. Figure 74 on the following page shows how this might be done for `operator+`. The code looks long but, when the compiler expands `Coordinate<double,3>`, it will see something like this: 242 243 244

  ```
  if (3 == 1)
      ....
  else if (3 == 2)
      ....
  else if (3 == 3)
      ....
  else
  ```

  Any reasonable compiler should be smart enough to compile the code for the "`3 == 3`" case and generate no code for the conditional expressions or the other cases.

  `<aside>` People sometimes wonder why a compiler should bother optimizing code such as

  ```
  if (1 == 0)
      ....
  ```

  on the grounds that no sane programmer would write code like this. Such optimizations are important, and very common, because a lot of code is ***not*** written explicitly by programmers, but is generated by template expansion, code generators, and in other ways. Unless the generator is very smart, generated code may be very stupid. `</aside>`

When we have obtained a type by applying a class template, we can do all of the usual things with it: 245

```
Coordinate<double, 3> c;            // 3D coordinate
Coordinate<double, 3> & rc;         // reference to a 3D coordinate
const Coordinate<double, 3> & rc;   // constant reference
                                    //    to a 3D coordinate
Coordinate<double, 3> * pc;         // pointer to a 3D coordinate
....
```

7.2.2 `class` or `typename`?

Early versions of C++ with templates used "`class`" where we have been using "`typename`". For example: 246

```
template<class T>
class Coordinate { ....
```

This usage suggested that the argument replacing `T` had to be a class and could not be a built-in type, such as `int`. The keyword `typename` suggests that ***any*** type can be used, including the built-in types.

```
friend Coordinate<Type, Dim> operator+(
      const Coordinate<Type, Dim> & left,
      const Coordinate<Type, Dim> & right )
{
    Coordinate<Type, Dim> result;
    if (Dim == 1)
    {
        result.c[0] = left.c[0] + right.c[0];
    }
    else if (Dim == 2)
    {
        result.c[0] = left.c[0] + right.c[0];
        result.c[1] = left.c[1] + right.c[1];
    }
    else if (Dim == 3)
    {
        result.c[0] = left.c[0] + right.c[0];
        result.c[1] = left.c[1] + right.c[1];
        result.c[2] = left.c[2] + right.c[2];
    }
    else
    {
        for (int d = 0; d != Dim; ++d)
            result.c[d] = left.c[d] + right.c[d];
    }
    return result;
}
```

Figure 74: A more elaborate version of `operator+`

Prefer* `typename` *to* `class` *for template parameters.

7.3 Compiling Template Code

We have seen that normal practice in C++ programming is to put declarations into header files and definitions into implementation files. This does not work for templates. The compiler cannot generate code for a function such as 247

```
template<typename T>
void Coordinate<T>::move(T dx, T dy)
{
    x += dx;
    y += dy;
}
```

without knowing the argument that replaces `T` and, if this definition is in an implementation file, the compiler cannot access it when it compiles a call such as `v.move(2,3)` (remember that implementation files must never depend on other implementation files, as discussed in Section 4.3 on page 72).

There are several solutions. Different platforms have different policies, but a solution that works on all platforms is to treat any template code, even a function such as `move` above, as a ***declaration***, and put it into a header file.

Put all template code into header files.

7.4 Template Return Types

It does not make sense to use a template type as a return type like this: 248

```
template<typename T>
T f()
{
    return ???
}
```

Even if we could write a sensible expression after `return`, the compiler could not deduce the template argument at the call site: In C++, it is not possible to overload a function based on the return type alone.

In general, if the return type of a function is a template parameter, then the function must have at least one parameter typed with the same template parameter, as in:

```
template<typename T>
T f(const T & param) { .... }
```

But even with a template parameter type, it can be difficult to figure out the actual return type. Consider a template function 249

```
template<typename T>
??? f(T t)
{
    return t.foo();
}
```

Since `foo` can return different types for different parameter types `T`, we cannot provide a fixed return type for this function.

In C++98, there is no solution for this. Since C++11, it is possible to ***automatically deduce*** the type, using the `auto` keyword. We have to make two changes to our template function: First, move the (still unknown) return type ***after*** the input type(s). This is another new C++11 feature, called ***trailing return type***, specified with the `->` operator. Second, let the compiler deduce the type of (here) `f.foo()` through the `decltype` keyword:

`auto`

`->`

`decltype`

```
template<typename T>
auto f(T t) -> decltype(t.foo())
{
    return t.foo();
}
```

It is also possible to declare new variables using `decltype` within the body of a function:

```
decltype(t.foo()) bar;      // make bar the same type as t.foo()
```

Of course, `decltype` and `auto` also work for non-template functions.

7.5 Template Specialization

It is sometimes necessary or desirable, for efficiency or other reasons, to provide a special implementation for some particular value of the template parameters. Suppose, for example, that we want to have both a general template class for all kinds of coordinates, as above, but we also want to give special treatment to three-dimensional coordinates with `double` elements. This
250 is called ***specialization***. Suppose the original class was

```
template<typename T>
class Widget
....
```

The specialized version will begin

```
template<>
class Widget<specType>
....
```

where `specType` is the value of `T` for which we are providing a specialized implementation. In the rest of the class, we must replace all occurrences of `<T>` with `specType`, including changing
251 `Widget<T>` to `Widget<specType>`.

252 253 254 Figure 75 on the facing page shows the result of specializing the generic coordinate class for 3D `double` coordinates. The constructor has been modified with the addition of an output statement to demonstrate that the specialized version is actually used. The definition

```
Coordinate<double,3> c;
```

produces the message

```
3D Coordinate
```

```
template<>
class Coordinate<double,3>
{
public:
    Coordinate<double,3>()
    {
        c[0] = 0; c[1] = 0; c[2] = 0;
        cout << "3D Coordinate" << endl;
    }

    double & operator[](int i)
    {
         assert(0 <= i && i < 3);
         return c[i];
    }

    friend Coordinate<double,3> operator+(
          const Coordinate<double,3> & left,
          const Coordinate<double,3> & right )
    {
        Coordinate<double,3> result;
        result.c[0] = left.c[0] + right.c[0];
        result.c[1] = left.c[1] + right.c[1];
        result.c[2] = left.c[2] + right.c[2];
        return result;
    }

    friend ostream & operator<<(ostream & os,
                                const Coordinate<double,3> & coord)
    {
        return os << '(' <<
            coord.c[0] << ", " <<
            coord.c[1] << ", " <<
            coord.c[2] << ')';
    }

private:
    double c[3];
};
```

Figure 75: A specialized version of class `Coordinate`

Within the declaration of the specialized version, we have unrolled the `for`-loops and made other small changes. We could also add functions, such as cross product, that are useful for 3D coordinates but not other coordinates.

It is also possible to ***partially specialize*** a template class with two or more template parameters. For example, we could specialize `Coordinate` to 3D coordinates with any element type. The
255 class declaration would begin

```
template<typename Type>
class Coordinate<Type,3>
....
```

Within the class declaration, instances of `Dim` are replaced by 3, but instances of `Type` are left unchanged. References to the class all have the form `Coordinate<Type,3>`.

7.6 Template Metaprogramming

Template specialization is the key to ***template metaprogramming***. The following class
256 declaration is allowed:

```
template<int N>
class Fac
{
public:
    static const int val = N * Fac<N-1>::val;
};
```

However, there is a problem: instantiating `Fac<4>` requires instantiating `Fac<3>` requires instantiating . . . . We can terminate the recursion by providing a specialized class `Fac<0>`, as follows:

```
template<>
class Fac<0>
{
public:
    static const int val = 1;
};
```

257 With these declarations, the following code returns 5040:

```
int n = Fac<7>::val;
```

Note that this computation is performed at *compile-time*, not at *run-time*! When the program is executed, the value 5040 has already been computed and is assigned like a constant. Like with other templates, this only works for constants or literals, since the value must be known at compile-time.

A consequence of template metaprogramming is that the compilation can take significantly longer, since now a potentially large amount of computations have to be performed by the compiler (think of the template metaprogram as a program that is running inside the compiler, leaving its results in the object code).

Unfortunately, it is generally not possible to look at the C++ code generated from a template, since this is handled internally in the compiler (rather than by a preprocessor, where you can inspect the results before compilation). However, you *can* look at the produced object code using a debugger or disassembler. On the x86 architecture, the following instructions[30] are generated for a function containing only the call to `int n = Fac<7>::val;`:

```
pushl %ebp
movl %esp, %ebp
subl $4, %esp
movl $5040, -4(%ebp)
leave
ret
```

here you can see the constant `$5040` computed by the template metaprogram. You've seen another example for template metaprogramming in the first lecture (Figure 2), where a metaprogram generated a parser for a given grammar at compile-time.

7.7 Default Arguments

Function declarations may have default arguments: 258

```
void foo(int n = 0) { .... }
```

The same is true of template class declarations. For example, if we changed the declaration of `Coordinate` in Figure 73 on page 140 to

```
template<typename Type = double, int Dim = 3>
class Coordinate
....
```

and defined

```
Coordinate<int> u;
Coordinate<> c;
```

then `u` would be a 3D coordinate of `int`s and `c` would be a 3D coordinate of `double`s.

Note that the definition

```
Coordinate c;
```

is ***not allowed***. The brackets `<>` are required even when we are using the default values of all parameters. (The same is true for functions, of course.)

[30]For details on x86 assembly syntax, you can consult the Wikibook "x86 Assembly": http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax.

References

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example*. Addison-Wesley.

Prata, S. (2012). *C++ Primer Plus* (6th ed.). Addison-Wesley.

Vandevoorde, D., N. M. Josuttis, and D. Gregor (2017). *C++ Templates* (2nd ed.). Addison-Wesley. http://tmplbook.com.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.