PREV
4. Classes

NEXT
6. Library Overview

5. Templates

Your quote here.

– B. Stroustrup

5.1. INTRODUCTION

Someone who wants a vector is unlikely always to want a vector of doubles. A vector is a general concept, independent of the notion of a floating-point number. Consequently, the element type of a vector ought to be represented independently. A *template* is a class or a function that we parameterize with a set of types or values. We use templates to represent concepts that are best understood as something very general from which we can generate specific types and functions by specifying arguments, such as the element type double.

5.2. PARAMETERIZED TYPES

We can generalize our vector-of-doubles type to a vector-of-anything type by making it a template and replacing the specific type double with a parameter. For example:

Click here to view code image

```
template<typename T>
class Vector {
private:
        T* elem;     //  elem points to an array of sz elements of type
        int sz;
public:
        explicit Vector(int s);            //  constructor: establish inva
        ˜Vector() { delete[] elem; }     //  destructor: release resource

        //  ... copy and move operations ...

        T& operator[](int i);
        const T& operator[](int i) const;
        int size() const { return sz; }
};
```

The template<typename T> prefix makes T a parameter of the declaration it prefixes. It is C++'s version of the mathematical "for all T" or more precisely "for all types T." Using class to introduce a type parameter is equivalent to using typename, and in older code we often see template<class T> as the prefix.

The member functions might be defined similarly:

Click here to view code image

```
template<typename T>
Vector<T>::Vector(int s)
{
        if (s<0)
                throw Negative_size{};
        elem = new T[s];
        sz = s;
}

template<typename T>
const T& Vector<T>::operator[](int i) const
{
        if (i<0 || size()<=i)
                throw out_of_range{"Vector::operator[]"};
        return elem[i];
}
```

Given these definitions, we can define Vectors like this:

Click here to view code image

```
Vector<char> vc(200);              // vector of 200 characters
Vector<string> vs(17);             // vector of 17 strings
Vector<list<int>> vli(45);         // vector of 45 lists of integers
```

The >> in Vector<list<int>> terminates the nested template arguments; it is not a misplaced input operator. It is not (as in C++98) necessary to place a space between the two >s.

We can use Vectors like this:

Click here to view code image

```
void write(const Vector<string>& vs)            // Vector of some s
{
         for (int i = 0; i!=vs.size(); ++i)
                cout << vs[i] << '\n';
}
```

To support the range-for loop for our Vector, we must define suitable begin() and end() functions:

Click here to view code image

```
template<typename T>
T* begin(Vector<T>& x)
{
        return x.size() ? &x[0] : nullptr;         // pointer to first elen
}

template<typename T>
T* end(Vector<T>& x)
{
        return begin(x)+x.size();        // pointer to one–past–last ele
}
```

Given those, we can write:

Click here to view code image

```
void f2(Vector<string>& vs)        // Vector of some strings
{
        for (auto& s : vs)
                cout << s << '\n';
}
```

Similarly, we can define lists, vectors, maps (that is, associative arrays), unordered maps (that is, hash tables), etc., as templates (Chapter 9).

Templates are a compile-time mechanism, so their use incurs no run-time overhead compared to hand-crafted code. In fact, the code generated for Vector<double> is identical to the code generated for the version of Vector from Chapter 4. Furthermore, the code generated for the standard-library vector<double> is likely to be better (because more effort has gone into its implementation).

In addition to type arguments, a template can take value arguments. For example:

Click here to view code image

```
template<typename T, int N>
struct Buffer {
        using value_type = T;
        constexpr int size() { return N; }
        T[N];
        // ...
};
```

The alias (value_type) and the constexpr function are provided to allow users (read-only) access to the template arguments.

Value arguments are useful in many contexts. For example, Buffer allows us to create arbitrarily sized buffers with no overheads from the use of free store (dynamic memory):

Click here to view code image

```
Buffer<char,1024> glob;   // global buffer of characters (statically

void fct()
{
        Buffer<int,10> buf; // local buffer of integers (on the stack)
        // ...
}
```

A template value argument must be a constant expression.

5.3. FUNCTION TEMPLATES

Templates have many more uses than simply parameterizing a container with an element type. In particular, they are extensively used for parameterization of both types and algorithms in the standard library (§9.6, §10.6). For example, we can write a function that calculates the sum of the element values of any container like this:

Click here to view code image

```
template<typename Container, typename Value>
Value sum(const Container& c, Value v)
{
        for (auto x : c)
```

```
                        v+=x;
                return v;
        }
```

The Value template argument and the function argument v are there to allow the caller to specify the type and initial value of the accumulator (the variable in which to accumulate the sum):

Click here to view code image

```
void user(Vector<int>& vi, std::list<double>& ld, std::vector<compl
{
        int x = sum(vi,0);                          // the sum of
        double d = sum(vi,0.0);                     // the sum of
        double dd = sum(ld,0.0);                    // the sum of
        auto z = sum(vc,complex<double>{0.0,0.0});      // the su
}
```

The point of adding ints in a double would be to gracefully handle a number larger than the largest int. Note how the types of the template arguments for sum<T,V> are deduced from the function arguments. Fortunately, we do not need to explicitly specify those types.

This sum() is a simplified version of the standard-library accumulate() (§12.3).

A function template can be a member function, but not a virtual member. The compiler would not know all instantiations of such a template in a program so it could not generate a vtbl (§4.4).

5.4. CONCEPTS AND GENERIC PROGRAMMING

What are templates for? In other words, what programming techniques are effective when you use templates? Templates offer:

• The ability to pass types (as well as values and templates) as arguments without loss of information. This implies excellent opportunities for inlining, of which current implementations take great advantage.

• Delayed type checking (done at instantiation time). This implies opportunities to weave together information from different contexts.

• The ability to pass constant values as arguments. This implies the ability to do compile-time computation.

In other words, templates provide a powerful mechanism for compile-time computation and type manipulation that can lead to very compact and efficient code. Remember that types (classes) can contain both code and values.

The first and most common use of templates is to support *generic programming*, that is, programming focused on the design, implementation, and use of general algorithms. Here, "general" means that an algorithm can be designed to accept a wide variety of types as long as they meet the algorithm's requirements on its arguments. The template is C++'s main support for generic programming. Templates provide (compile-time) parametric polymorphism.

Consider the `sum()` from §5.3. It can be invoked for any data structure that supports `begin()` and `end()` so that the range-`for` will work. Such structures include the standard-library `vector`, `list`, and `map`. Furthermore, the element type of the data structure is limited only by its use: it must be a type that we can add to the `Value` argument. Examples are `int`s, `double`s, and `Matrix`es (for any reasonable definition of `Matrix`). We could say that the `sum()` algorithm is generic in two dimensions: the type of the data structure used to store elements ("the container") and the type of elements.

So, `sum()` requires that its first template argument is some kind of container and its second template argument is some kind of number. We call such requirements *concepts*. Unfortunately, concepts cannot be expressed directly in C++11. All we can say is that the template argument for `sum()` must be types. There are techniques for checking concepts and proposals for direct language support for concepts [Stroustrup,2013] [Sutton,2012], but both are beyond the scope of this thin book.

Good, useful concepts are fundamental and are discovered more than they are designed. Examples are integer and floating-point number (as defined even in Classic C), more general mathematical concepts such as field and vector space, and container. They represent the fundamental concepts of a field of application. Identifying and formalizing to the degree necessary for effective generic programming can be a challenge.

For basic use, consider the concept *Regular*. A type is regular when it behaves much like an `int` or a `vector`. An object of a regular type

• can be default constructed.

• can be copied (with the usual semantics of copy yielding two objects that are independent and compare equal) using a constructor or an assignment.

• can be compared using == and !=.

• doesn't suffer technical problems from overly clever programming tricks.

A string is another example of a regular type. Like int, string is also *Ordered*. That is, two strings can be compared using <, <=, >, and >= with the appropriate semantics. Concepts is not just a syntactic notion, it is fundamentally about semantics. For example, don't define + to divide; that would not match the requirements for any reasonable number.

5.5. FUNCTION OBJECTS

One particularly useful kind of template is the *function object* (sometimes called a *functor*), which is used to define objects that can be called like functions. For example:

Click here to view code image

```
template<typename T>
class Less_than {
        const T val;    // value to compare against
public:
        Less_than(const T& v) :val(v) { }
        bool operator()(const T& x) const { return x<val; } // call op
};
```

The function called operator() implements the "function call," "call," or "application" operator ().

We can define named variables of type Less_than for some argument type:

Click here to view code image

```
Less_than<int> lti {42};              // lti(i) will compare i to 42 usi
Less_than<string> lts {"Backus"}; // lts(s) will compare s to "Back
```

We can call such an object, just as we call a function:

Click here to view code image

```
void fct(int n, const string & s)
{
        bool b1 = lti(n);      //  true if n<42
        bool b2 = lts(s);      //  true if s<"Backus"
        //  ...
}
```

Such function objects are widely used as arguments to algorithms. For example, we can count the occurrences of values for which a predicate returns true:

Click here to view code image

```
template<typename C, typename P>
int count(const C& c, P pred)
{
        int cnt = 0;
        for (const auto& x : c)
                if (pred(x))
                        ++cnt;
        return cnt;
}
```

A *predicate* is something that we can invoke to return true or false. For example:

Click here to view code image

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const
{
        cout << "number of values less than " << x
                << ": " << count(vec,Less_than<int>{x})
                << '\n';
        cout << "number of values less than " << s
                << ": " << count(lst,Less_than<string>{s})
                << '\n';
}
```

Here, Less_than<int>{x} constructs an object for which the call operator compares to the int called x; Less_than<string>{s} constructs an object that compares to the string called s. The beauty of these function objects is that they carry the value to be compared against with them. We don't have to write a separate function for each value (and each type), and we don't have to introduce nasty global variables to hold values. Also, for a simple function object like Less_than inlining is simple, so that a call of Less_than is far more efficient than an indirect function call. The ability to carry data plus their efficiency make function objects particularly useful as arguments to algorithms.

Function objects used to specify the meaning of key operations of a general algorithm (such as Less_than for count()) are often referred to as *policy objects*.

We have to define Less_than separately from its use. That could be seen as inconvenient. Consequently, there is a notation for implicitly generating function objects:

Click here to view code image

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const
{
        cout << "number of values less than " << x
                << ": " << count(vec,[&](int a){ return a<x; })
                << '\n';
        cout << "number of values less than " << s
                << ": " << count(lst,[&](const string& a){ return a<s; })
                << '\n';
}
```

The notation [&](int a){ return a<x; } is called a *lambda expression*. It generates a function object exactly like Less_than<int>{x}. The [&] is a *capture list* specifying that local names used (such as x) will be accessed through references. Had we wanted to "capture" only x, we could have said so: [&x]. Had we wanted to give the generated object a copy of x, we could have said so: [=x]. Capture nothing is [ ], capture all local names used by reference is [&], and capture all local names used by value is [=].

Using lambdas can be convenient and terse, but also obscure. For nontrivial actions (say, more than a simple expression), I prefer to name the operation so as to more clearly state its purpose and to make it available for use in several places in a program.

In §4.5.4, we noted the annoyance of having to write many functions to perform operations on elements of vectors of pointers and unique_ptrs, such as draw_all() and rotate_all(). Function objects (in particular, lambdas) can help by allowing us to separate the traversal of the container from the specification of what is to be done with each element.

First, we need a function that applies an operation to each object pointed to by the elements of a container of pointers:

Click here to view code image

```
template<typename C, typename Oper>
void for_all(C& c, Oper op)       // assume that C is a container of
{
        for (auto& x : c)
                op(*x);           // pass op() a reference to each ele
}
```

Now, we can write a version of user() from §4.5 without writing a set of _all functions:

Click here to view code image

```
void user()
{
        vector<unique_ptr<Shape>> v;
        while (cin)
                v.push_back(read_shape(cin));
        for_all(v,[](Shape& s){ s.draw(); });          // draw_all()
        for_all(v,[](Shape& s){ s.rotate(45); });    // rotate_all(45)
}
```

I pass a reference to Shape to a lambda so that the lambda doesn't have to care exactly how the objects are stored in the container. In particular, those for_all() calls would still work if I changed v to a vector<Shape *>.

5.6. VARIADIC TEMPLATES

A template can be defined to accept an arbitrary number of arguments of arbitrary types. Such a template is called a *variadic template*. For example:

Click here to view code image

```
void f() { }        //  do nothing

template<typename T, typename... Tail>
void f(T head, Tail... tail)
{
        g(head);       //  do something to head
        f(tail...); //  try again with tail
}
```

The key to implementing a variadic template is to note that when you pass a list of arguments to it, you can separate the first argument from the rest. Here, we do something to the first argument (the head) and then recursively call f() with the rest of the arguments (the tail). The ellipsis, ..., is used to indicate "the rest" of a list. Eventually, of course, tail will become empty and we need a separate function to deal with that.

We can call this f() like this:

```
int main()
{
        cout << "first: ";
        f(1,2.2,"hello");

        cout << "\nsecond: ";
        f(0.2,'c',"yuck!",0,1,2);
        cout << "\n";
}
```

This would call f(1,2.2,"hello"), which will call f(2.2,"hello"), which will call f("hello"), which will call f(). What might the call g(head) do? Obviously, in a real program it will do whatever we wanted done to each argument. For example, we could make it write its argument (here, head) to output:

```
template<typename T>
void g(T x)
{
        cout << x << " ";
}
```

Given that, the output will be:

```
first: 1 2.2 hello
second: 0.2 c yuck! 0 1 2
```

It seems that `f()` is a simple variant of `printf()` printing arbitrary lists or values – implemented in three lines of code plus their surrounding declarations.

The strength of variadic templates (sometimes just called *variadics*) is that they can accept any arguments you care to give them. The weakness is that the type checking of the interface is a possibly elaborate template program.

Because of their flexibility, variadic templates are widely used in the standard library.

5.7. ALIASES

Surprisingly often, it is useful to introduce a synonym for a type or a template. For example, the standard header `<cstddef>` contains a definition of the alias `size_t`, maybe:

```
using size_t = unsigned int;
```

The actual type named `size_t` is implementation-dependent, so in another implementation `size_t` may be an `unsigned long`. Having the alias `size_t` allows the programmer to write portable code.

It is very common for a parameterized type to provide an alias for types related to their template arguments. For example:

```
template<typename T>
class Vector {
public:
        using value_type = T;
        // ...
};
```

In fact, every standard-library container provides value_type as the name of its value type (Chapter 9). This allows us to write code that will work for every container that follows this convention. For example:

Click here to view code image

```
template<typename C>
using Element_type = typename C::value_type;    // the type of C's

template<typename Container>
void algo(Container& c)
{
        Vector<Element_type<Container>> vec;        // keep result
        // ...
}
```

The aliasing mechanism can be used to define a new template by binding some or all template arguments. For example:

Click here to view code image

```
template<typename Key, typename Value>
class Map {
        // ...
};

template<typename Value>
using String_map = Map<string,Value>;

String_map<int> m;        // m is a Map<string,int>
```

5.8. TEMPLATE COMPILATION MODEL

The type checking provided for templates checks the use of arguments in the template definition rather than against an explicit interface (in a template declaration). This provides a compile-time variant of what is often called *duck typing* ("If it walks like a duck and it quacks like a duck, it's a duck"). Or – using more technical terminology – we operate on values, and the presence and meaning of an operation depend solely on its operand values. This differs from the alternative view that objects have types, which determine the presence and meaning of operations. Values "live" in objects. This is the way objects (e.g., variables) work in C++, and only values that meet an object's requirements can be put into it. What is done at compile time using templates does not involve objects, only values.

The practical effect of this is that to use a template, its definition (not just its declaration) must be in scope. For example, the standard header `<vector>` holds the definition of `vector`. An unfortunate side effect is that a type error can be found uncomfortably late in the compilation process and can yield spectacularly bad error messages because the compiler found the problem by combining information from several places in the program.

5.9. ADVICE

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapters 20-29 of [Stroustrup,2013].

[2] Use templates to express algorithms that apply to many argument types; §5.1.

[3] Use templates to express containers; §5.2.

[4] Use templates to raise the level of abstraction of code; §5.2.

[5] When defining a template, first design and debug a non-template version; later generalize by adding parameters.

[6] Templates are type-safe, but checking happens too late; §5.4.

[7] A template can pass argument types without loss of information.

[8] Use function templates to deduce class template argument types; §5.3.

[9] Templates provide a general mechanism for compile-time programming; §5.4.

[10] When designing a template, carefully consider the concepts (requirements) assumed for its template arguments; §5.4.

[11] Use concepts as a design tool; §5.4.

[12] Use function objects as arguments to algoritms; §5.5.

[13] Use a lambda if you need a simple function object in one place only; §5.5.

[14] A virtual function member cannot be a template member function.

[15] Use template aliases to simplify notation and hide implementation details; §5.7.

[16] Use variadic templates when you need a function that takes a variable number of arguments of a variety of types; §5.6.

[17] Don't use variadic templates for homogeneous argument lists (prefer initializer lists for that); §5.6.

[18] To use a template, make sure its definition (not just its declaration) is in scope; §5.8.

[19] Templates offer compile-time "duck typing"; §5.8.

[20] There is no separate compilation of templates: #include template definitions in every translation unit that uses them.