

TCC CSCI-2843 C++ Programming Language

Lecture 11 Notes

Function Templates

Earlier, we introduced class templates that allowed us to write types that were applicable to a wide variety of types through type parameters. Along these same lines, we can also make **function templates** that are used to generate functions that vary only on the types that they operate on.

Consider the function

```
void push_ints(vector<int>& container) {
    for (int i = 0; i < 10; ++i) {
        container.push_back(i);
    }
}
```

This function is wired to work only with `vector<int>`, but what if we want to populate a `list<int>` the same way? We can write (yet) another function

```
void push_ints(list<int>& container) {
    for (int i = 0; i < 10; ++i) {
        container.push_back(i);
    }
}
```

but that means twice the work when the innards of the function are **the same** and the only difference in the function declarations themselves is the **type(s)** it operates on. This is similar to the motivation for creating **class templates**, so C++ gives us function templates to solve the problem in a similar fashion:

```
template <typename Container>
void push_ints(Container& container) {
    for (int i = 0; i < 10; ++i) {
        container.push_back(i);
    }
}
```

We can now use this function template to generate a function that operates on a `vector<int>` with

```
vector<int> v;

push_ints<vector<int>> >(v);
```

which will populate `v`. Note the space between the two `>` characters – this is required because current compilers would see two `>` characters together as the operator `>>` and not as separate `>` tokens, thus giving an error. The next version of the standard fixes this syntactic problem.

We can the turn around and with the **same template**, generate a function that operates on `list<int>`:

```
list<int> g;

push_ints<list<int>> >(g);
```

We use the same function template parameterized with a different type to generate a different function (that you do not ever see). This allows common operations to be parameterized and applied to a wider variety of types without having to provide explicit overloads for each type.

Inference

Note that we used **explicit** parameterization of the function template in our previous example, meaning that we put the type we wanted the type parameter to be in `<>` brackets after the function name. This is one way to absolutely specify type parameters of a function template, just like we do when we parameterize class templates.

As a convenience, as long as the compiler can **infer** the type parameters **solely** from the arguments passed to a function template usage, we can let the type parameters be **implicit**. As an example:

```
vector<int> v;
```

```
push_ints(v);
```

Here, we do **not** provide explicit template parameters, but use the function template just like a function, i.e. we call it. The compiler will note that we are not **really** calling a function, but recognizes that we **are** using a function template and will attempt to generate a function that fits our scenario using the arguments we provide.

It first notes that the first (and only) argument of `push_ints` uses the type parameter, `Container`. It then notes that we have passed a `vector<int>` as the first argument. It then infers that the type of `Container` **must** be `vector<int>`. It will continue in this fashion with all arguments, trying to infer type parameters until no more arguments are left. If all type parameters have been inferred and if there are no conflicts in the types passed in with those expected, then the appropriate function is generated and a **real** function call is placed in the code.

Remember that for this to work **all** type parameters must be inferred from the arguments passed to the function call, consistently and unambiguously. A case where there is a conflict would be

```
template <typename T>
void an_error(T a, T b) { ... }
```

```
string x;
int i;
```

```
an_error(x, i);
```

Here, the type parameter, `T`, will be inferred **first** as `string` because the first function argument passed in, `x`, is of type `string`. Then, the second argument has an `int` passed in, but since `T` has already been inferred as `string`, there is a type conflict – you cannot pass an `int` where a `string` is expected.

We need to still explicitly parameterize a function template in those cases where not all type parameters may be inferred by the arguments. An example

```
template <typename From, typename To>
To round(From from) { return To(from + .5); }
```

In this example, we could not just write

```
double x = 1.2;
```

```
int i = round(x); // compiler error – cannot deduce template parameter, To
```

because the `To` type parameter cannot be inferred from the arguments of the function (the return type of a function is **not** used in overloading **or** type parameter inference). This requires us to write

```
double x = 1.2;
```

```
int i = round<double, int>(x);
```

which is not too terrible, since it documents quite nicely that we are rounding a **double** into an **int**.

Parametric Polymorphism

When we first learned about **polymorphism** we stated that it is where objects share a common interface, but provide (potentially) distinct behaviors. The mechanism we used to implement this concept was through **inheritance** coupled with **virtual functions** that we overload in **derived classes**.

Consider this

```
template <typename T>
void call_f(T& t) {
    t.f();
}
```

```
class X {
public:
    void f() { ... }
```

```

};

class Y {
public:
    int f() { ... }
};

int main() {
    X an_x;
    Y a_y;

    call_f(an_x);
    call_f(a_y);

    return 0;
}

```

Note that we use the function template to indirectly invoke a function named, `f`, on whatever object is passed in. We construct an `X` object and a `Y` object that are then passed into calls to `call_f`. Both of these are perfectly valid and will result in their respective `f` functions being called, even though the classes are **not** related through inheritance and do **not** provide virtual functions, let alone have the same function signatures. The function template will generate the correct function call regardless. This is another example of two objects sharing a **common interface**, but providing **distinct behavior**. In other words – another form of polymorphism.

Because this form does not use inheritance and virtual functions, but instead relies on **parameterization** to generate the correct calls, we can call it **parametric polymorphism**. Note that this function resolution happens at **compile-time** instead of **run-time** as in the case of traditional polymorphism. For this reason, it is also called **compile-time polymorphism** and the use of virtual functions to gain polymorphism, **run-time polymorphism**.

Compile-time polymorphism can save us from the overhead of virtual function dispatch at run-time, thus possibly generating faster code. However, the real speed gains are that, since the compiler sees all the source code that is being plugged into a function template instantiation, it is capable of **inlining** code while it generates the functions from the function templates. This can have an even greater boost in performance and code economy.

Generic Algorithms

Consider the common case of linearly searching through a container for a given item. We can write this using `vector<int>` and its iterators thusly

```

vector<int>::iterator find(vector<int>::iterator a, vector<int>::iterator b, const int& value) {
    for (; a != b && *a != value; ++a) {}
    return a;
}

```

This algorithm searches for a value using the iterator, `a`, that is the beginning of a range of values up to **but not including** the end of the range denoted by `b`. If the algorithm finds the value in the vector, it will return the iterator referencing that item. If it goes all the way through the vector without finding the value, it will return an iterator equivalent to the end of the range (i.e. `b`). We can use this like

```

vector<int> v;
push_ints(v);

vector<int>::iterator i = find(v.begin(), v.end(), 5);
if (i == v.end()) {
    cout << "Not found\n";
} else {
    cout << "Found\n";
}

```

This practice of using an iterator range of the form [a, b) where b is considered “one past the end” of the range, is common. This allows the b iterator to be used to denote “not found” as in this case.

The above algorithm is fine if we only needed to search for **ints**. We could incrementally improve this code by parameterizing the type the vector can hold:

```
template <typename T>
vector<T>::iterator find(vector<T>::iterator a, vector<T>::iterator b, const T& value) {
    for (; a != b && *a != value; ++a) {

    }
    return a;
}
```

Now we can search through `vector<string>`, `vector<float>`, etc.:

```
vector<string> v;
... push stuff into v

if (find(v.begin(), v.end(), "some value") == v.end()) {
    ...
}
```

Now, `find` is a function **template** that will **infer** `T` to be `string` (yes the compiler is **that** smart) and will happily search for our given value without problem.

This **is** an improvement, but we note that we also may want search through lists. Do we write yet another function or function template? No – we just make our function template even **more generic**.

```
template <typename Iterator, typename T>
Iterator find(Iterator a, Iterator b, const T& value) {
    for (; a != b && *a != value; ++a) {

    }
    return a;
}
```

Now we can use `find` like

```
list<int> g;
push_ints(g);

if (find(g.begin(), g.end(), 5) == g.end()) {
    ...
}
```

as well as in our initial example of `vector<int>` or our other example of `vector<string>`. This is made possible by the fact that **all** iterators share a common interface – that of the **forward iterator** category, that allows them to be operated on (parametrically) **polymorphically**. The `find` algorithm only uses the interface required by a forward iterator, therefore just about any iterator type may be used in conjunction with it – including **pointers/arrays**.

```
int a[] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };

int* p = find(a, a + 10, 8);

if (p == a + 10) {
    cout << "not found\n";
}
```

The fact that all iterators were modeled on the semantics of pointers means that in **most** circumstances, we can use a pointer everywhere an iterator is required.

Standard Generic Algorithms

The `find` function we developed earlier is already in the standard library. The C++ standard provides a plethora of function templates that use containers in conjunction with the iterator model to perform many common algorithms on as wide a variety of types possible, all thanks to the template mechanism for classes and functions. An incomplete list of some of these algorithms is given below:

<p>Non-mutating Algorithms</p> <ul style="list-style-type: none"> adjacent_find binary_search count count_if equal find find_end find_first_of find_if for_each mismatch search search_n <p>Sorting</p> <ul style="list-style-type: none"> binary_search equal_range inplace_merge is_sorted lexicographical_compare lower_bound merge next_permutation nth_element partial_sort partial_sort_copy prev_permutation sort stable_sort upper_bound 	<p>Numeric Algorithms</p> <ul style="list-style-type: none"> accumulate adjacent_difference inner_product negate partial_sum <p>Set Operations</p> <ul style="list-style-type: none"> includes set_difference set_intersection set_symmetric_difference set_union <p>Heap Operations</p> <ul style="list-style-type: none"> make_heap pop_heap push_heap sort_heap is_heap <p>Minimum and Maximum</p> <ul style="list-style-type: none"> max max_element min min_element
--	---

These are all accessed from the header, `<algorithm>` and are all in the `std` namespace.

Function Objects

Many of the algorithms introduced in the last section can use **function objects** to manipulate data or to make decisions during the course of the algorithm's processing.

The operator()

To make an **object** act like a **function**, we can overload the **operator()**. This operator is unique in that it can be defined with however many arguments we want, including zero. Consider the class

```
class F {
private:
    int value;
public:
    F(int init_value) : value(init_value) {}
    int operator()() const { return value; }
};
```

This can be used like

```
F f(10);

cout << f() << '\n';
int i = f();
```

First, we construct the object, `f`, with a value it will hold on to. Then we can apply the `operator()` anytime we wish and it will happily return us whatever value we constructed it with. Remember that even though the expression:

```
f()
```

looks like we are invoking a function directly, what we are **really** doing is invoking an object's `operator()` that takes no arguments.

A more useful example is

```
class iota {
private:
    int i;
public:
    iota(int start = 0) : i(start) { }
    int operator() { return i++; }
};
```

This implements a simplistic **sequence generator** (often referred to as **iota** in some circles) that can be used like

```
iota g(100);

for (int i = 0; i < 100; ++i) {
    cout << g() << endl;
}
```

which will output the numbers 100..199 to `cout`.

Since we can take in however many arguments we want, we can do something more clever (and interesting)

```
class add_to {
private:
    int offset;
public:
    add_to(int init_offset) : offset(init_offset) { }
    int operator()(int value) const { return value + offset; }
};
```

We can use this like

```
add_to f(10);

for (int i = 0; i < 10; ++i) {
    cout << f(i) << endl;
}
```

OK – not so interesting, but what it **does** show is that function objects can hold **state** in between invocations of their `operator()` which is something that ordinary functions **cannot do**. We can use this to great advantage when used in conjunction with generic algorithms.

Using Function Objects

Consider the `find` function

```
template <typename Iterator, typename T>
Iterator find(Iterator a, Iterator b, const T& value) {
    for (; a != b && *a != value; ++a) { }
    return a;
}
```

In this algorithm, a value is provided as an argument and a hard-coded decision of `!=` is used to determine if that value is found. What if we wanted to search by some other criteria? We could pass in a function

object that implements the decision with an interface general enough to allow almost any kind of searching. All we need to do is provide a `find_if` version of this algorithm:

```
template <typename Iterator, typename Predicate>
Iterator find(Iterator a, Iterator b, Predicate test) {
    for (; a != b && test(*a); ++a) {}
    return a;
}
```

Note that we use the object passed in as test like a function – that is, we apply the `operator()` to it. All that we require is that the function object’s `operator()` takes a single argument that is type compatible with the type of object the iterator is referencing. We can code the equality test to get the same functionality as our `find` function:

```
class equal_to {
private:
    int value;
public:
    bool operator()(int arg) const { return value == arg; }
};
```

This can be used like

```
vector<int> g;
push_ints(g);

if (find_if(g.begin(), g.end(), equal_to(5)) == g.end()) {
    ...
}
```

The type for `Predicate` in `find_if` will be inferred as `equal_to`, and the anonymously constructed `equal_to` object we pass in will be (trivially) copied in as the test argument. From there, the body of the `find_if` function will test the return value of `test(*a)` to determine if the condition is satisfied. The condition, in this case, is true only if the value passed in (what the iterator is referencing) is equal to whatever the `equal_to` object was constructed with (5).

To see the flexibility of this we can write

```
class is_divisible_by {
private:
    int value;
public:
    is_divisible_by(int init_value) : value(init_value) {}
    bool operator()(int arg) const { return arg % value == 0; }
};
```

So we can write

```
if (find_if(g.begin(), g.end(), is_divisible_by(5)) == g.end()) {
    ...
}
```

which will find the first item in the container that is divisible by 5.

Function Object Categories

In the previous examples, we’ve seen function objects that take zero or one argument. Some function objects we’ve used so far have been used to make decisions. In order to be able to document and differentiate what kinds of function objects are necessary, we need to categorize them.

Any function object that is used to make a decision is technically called a **predicate object**. “Predicate” is used to denote “conditional” as in the statement, “a good grade is predicated upon studying hard”.

The other aspect of function (and predicate) objects is how many arguments they take:

- Zero arguments – called a **generator** (sometimes **nullary**).
- One argument – called a **unary** function object.
- Two arguments – called a **binary** function object.

- More than two – use the associated Latin prefix along with “-ary” suffix.

Generators

To give you an example of how linked function objects are with generic algorithms, let’s look at an algorithm that can generate a sequence and place them in a container:

```
template <typename Iterator, typename Size, typename G>
void generate_n(Iterator a, Size n, G g) {
    for (; n > 0; --n, ++a) { *a = g(); }
}
```

This will iterate n times, calling the generator, g , for each iteration, placing the generated value into wherever the iterator, a , references, and increments a to refer to the next item. We can use it like this

```
int an_array[10];
generate_n(an_array, 10, iota(0));
```

which will fill `an_array` with the integers, 0..9. Note though, that we cannot directly use this with an **empty container**:

```
vector<int> v;
generate_n(v.begin(), 10, iota(0)); // oops – crash at run-time
```

This is because the `generate_n` algorithm **assumes** that the iterator is iterating over **already existing** items in a container. In our array example, the integers exist when the array is allocated, but with a vector, they don’t automatically get allocated, usually requiring a **push_back** to insert an item onto the end of the vector. With an empty vector, we cannot simply pass the “begin” iterator in, because it will be equal to the “end” iterator, which is invalid to dereference.

We do not want to try to `push_back` “fake” values because we don’t want to go through the initialization twice, which may be nontrivial for other types. What we need is some kind of adaptor that acts like an iterator, but performs a `push_back` when accessed.

Insertion Iterators

Insertion iterators allow us to adhere to the iterator interface, while performing insertions into containers instead of merely accessing existing items. The insertion iterator we will be looking at (and probably the most-used) is a **back-insertion iterator**, that (as its name implies) performs a `push_back`. The others, **front-insertion iterator**, and **insertion iterator** are left for future study.

We can construct a back-insertion iterator through the convenience function, `std::back_inserter`, that we get from the header, `<iterator>`. Its usage is as follows

```
vector<int> v;
generate_n(std::back_inserter(v), 10, iota(0)); // pushes the first ten integers onto the vector
```

In this case, the call, `std::back_inserter(v)`, constructs and returns a back-insertion iterator of the type

```
std::back_insert_iterator<vector<int>> >
```

The object of this type that is constructed is then passed into `generate_n` as the iterator type parameter, which is in turn inferred as this type. Now when the body of the `generate_n` function is executed, the statement sequence

```
for (; n > 0; --n, ++a) { *a = g(); }
```

will cause the back-insertion iterator to perform a `push_back` when the dereference and assignment are performed (through the overload of these operators on the back-insertion iterator).

Examples

Back-insertion iterators are very useful when used in conjunction with the standard fill, copy, and transform algorithms. I will give you hypothetical versions of the `fill_n` and `copy` functions and their usage and let you map out how it works.

Fill (Counted)

The `fill_n` algorithm can fill an iterator range with n copies of a value:

```
template <typename Iterator, typename Size, typename T>
void fill_n(Iterator out, Size n, const T& value) {
    for (; n > 0; --n, ++out) { *out = value; }
}
```

We can use it like

```
#include <algorithm>
#include <string>
#include <iterator>
#include <iostream>

int main() {
    std::string s;
    std::fill_n(std::back_inserter(s), 20, 'x')

    std::cout << s << '\n';

    return 0;
}
```

This is a somewhat wasteful (but instructive) way to construct a string with 20 'x's and outputs the string to the console.

Copy

The copy algorithm looks like

```
template <typename Input, typename Output >
void copy(Input a, Input b, Output out) {
    for (; a != b; ++a, ++out) { *out = *a; }
}
```

We can use it like

```
#include <algorithm>
#include <vector>
#include <list>
#include <iterator>

int main() {
    std::vector<int> v;
    std::list<double> g;

    std::generate_n(std::back_inserter(v), 100, iota(0));
    std::copy(v.begin(), v.end(), std::back_inserter(g));

    return 0;
}
```

This pushes the first 100 integers into the vector, `v` and then copies the contents into the list, `g`. Note that `g` holds **doubles**, but because of implicit conversions, the algorithm handles this copy without incident. If we flipped the copy (**doubles** into **ints**), we might get warnings from the generation of the copy function because of losing precision.

Transform

The transform algorithm is a little trickier, but still straightforward:

```
template <typename Input, typename Output, typename F>
void transform(Input a, Input b, Output out, F f) {
```

```

    for (; a != b; ++a, ++out) { *out = f(*a); }
}

```

Here, a unary function object may be passed in to transform items from one iterator range and the result output to another iterator. We can use this like

```

#include <vector>
#include <list>
#include <iterator>

int main() {
    std::vector<int> v;
    std::list<int> g;

    std::generate_n(std::back_inserter(v), 100, iota(0));
    std::transform(v.begin(), v.end(), std::back_inserter(g), add_to(10));

    return 0;
}

```

This will generate the first 100 integers into `v` and then copy these values into the list, `g`, while adding 10 to each one.

Containers

To round out our high-level overview of the standard library, below is a list of the containers provided:

<p>Random Access Containers</p> <ul style="list-style-type: none"> array[*] vector valarray string bitset <p>Lists</p> <ul style="list-style-type: none"> list forward_list[*] <p>Queues and Stacks</p> <ul style="list-style-type: none"> queue[†] deque priority_queue stack[†] 	<p>Associative Containers</p> <ul style="list-style-type: none"> map multimap unordered_map[‡] unordered_multimap[‡] set multiset unordered_set[‡] unordered_multiset[‡] <p>Data Structures</p> <ul style="list-style-type: none"> pair[*] tuple[*]
<p>[*] new to C++0X [†] container adapter [‡] “unordered” implies “using a hashing method”</p>	