Templates

C++ provides a mechanism to facilitate writing generic functions, classes, and structs - templates. Templates allow the author to define type variables which are evaluated and expanded at compile time. There are two distinct activities associated with templates - authoring and using. We will go over both. You have already used templates a bit in this course. The std::vector is a template which declares a single type variable, which gets "filled in" when used. For example:

```
std::vector<int> nums;
```

The `<int>` part provides the vector template with the specific type we are interested in using. The compiler sees this and constructs a vector specific to the `int` type. Using templates is pretty straight forward, as you can see. Templates may be used to define classes, structs, and even free functions.

Using Templates

C++ ships with a very powerful template library, called the STL - the Standard Template Library. We have already seen a couple of templates from that library - vector and string. We are going to look a little more deeply.

FYI, a good reference site for the stl, and much more is cplussplus.com

A word about the STL

The STL introduced a number of important concepts into C++. One of them is the notion of an `iterator`. In C++, an iterator is just an object which has the ability to iterate through the range of a container.

Using an iterator is very similar to using a pointer. It will feel the same. You use the `*` operator on an iterator to access its value. However, the operations you can perform will be constrained by the type of iterator. However, iterators are more complex beasts which fall into a classification, which describes their functionality:

- input / output (sequential single pass input)
- forward (input + forward iteration. all containers support forward iteration)
- bidirectional (like forward iterators but can also iterate backwards)
- random access (like forward but can access randomly)

Retrieving and Using an Iterator

So, how do you get an iterator to begin with? You can call container method(s) to retrieve one. For sequential iteration, supporting containers provide `begin` and `end` methods. `begin` returns an iterator to the first element in a container. `end` returns an iterator to one past the last element of a container. Iterator values are generally accessed using pointer notation:

```
string foo= "foo";
auto ifoo = foo.begin();
cout << *ifoo << endl;
```

You can safely use arithmetic on an iterator to advance it:

```
ifoo++;
cout << *ifoo << endl;
```

The common idiom for iterating, at least before c++11, was to do this:

```
for(auto i=foo.begin(); i != foo.end(); foo++) {
    cout << foo;
}
```

Notice the `i != foo.end()` test. End iterators return the value *after* the last container element. They are *only* useful for testing iteration. You should not access the value of an end iterator. Bad things will happen. You have been warned.
Iterators are commonly used in place of ints when iterating through a collection:

```
#include <vector>
#include <iostream>
using namespace std;
//
int main() {
    vector<string> foo = {"this","is","it"};
    for( auto i = foo.begin(); i != foo.end(); foo++) {
        cout << *i << endl;
    }

return 0;
}
```

String

Just as in Python, in C++, strings are containers. They are iterable, the support random access, as well as appending to their end. In fact, std::string is actually a typedef. It is defined thusly:

```
typedef string basic_string<char>
```

String Member Functions

Iterators

- begin
- end
- rbegin
- rend
- cbegin (c++11)
- cend (c++11)
- crbegin (c++11)
- crend (c++11)

Capacity

- size
- length
- max_size
- resize
- capacity
- reserve
- clear
- empty
- shrink_to_fit (c++11)

Element Access

- operator[]
- at
- back
- front

Modifiers

- operator+=
- append
- push_back
- assign
- insert
- erase
- replace
- swap
- pop_back

String Operations

- c_str
- data
- get_allocator
- copy
- find
- rfind
- find_first_of
- find_last_of
- find_first_not_of
- find_last_not_of
- substr
- compare

Vector

Iterators

- begin
- end
- rbegin
- rend
- cbegin
- cend
- crbegin
- crend

Access

- at
- operator[]
- front
- back
- data

Capacity

- empty
- size
- max_size
- reserve
- capacity
- shrink_to_fit

Modifiers

- clear
- insert
- emplace (c++11)
- erase
- push_back
- emplace_back (c++11)
- pop_back
- resize
- swap

Writing Templates

Before we begin, I want to discuss one limitation of templates, so you don't gt tripped up later: templates may not be implemented in cpp files; they must be written in header files. So keep that in mind in your future endeavors.

Templates begin with the `template` keyword, followed by an open angle bracket (`<`). For each type variable you want, you add `typename` followed by the variable name. If you provide more than one type variable, each must be separated by a comma. At the tail end, you close the affair with a close angle bracket (`>`).

Confusingly, the `class` property may be used in place of `typename`. In the template context, this has nothing to do with C++ classes. Because of this ambiguity, I prefer to use `typename` as it is clearer. Ok. Here is an example declaring two type variables in a template:

```
template <typename T, typename G>
```

By convention, T is used frequently as a typename variable. However, like all variables, this is arbitrary. Also note that conventionally, the type is capitalized.

Please note that modifiers such as &, *, and const are not part of the variable name. They appear in the usage.

The rest of the template is no different than a non-template declaration, except that one uses the variable in lieu of an actual variable. For example, here is a silly class template to store a secret:

“‘

template
class Secret {
T secret;
public:
Secret(const T& rhs) {
secret = rhs;
};

```
void share() const {
    cout << "The secret is " << secret << endl;
}

const T& getSecret() const {return secret;};

}
```
‘‘‘

We can use this template with a variety of data types. Lets give it a go:

(NOTE: from now on I am not going to bother with providing fully working code. Fragments should do fine. You should be able to apply them appropriately by now.)

```
cout << "Secret<string>" << endl;
Secret<string> strSecret("foobar");
strSecret.share();
// lets get the secret value
const auto& str_secret = strSecret.getSecret();
cout << "fetched secret " << str_secret << endl;
// this wouldn't compile
// str_secret += "foo";
cout << endl << "Secret<int>" << endl;
Secret<int> intSecret(42);
intSecret.share();
```

Likewise, implementing template functions is equally simple. Here is an example `max` which will work with any type that implements a `>` operator. This can be any built in type or user defined type; it doesn't matter as long as it meets the prerequisites. One more thing: the STL provides a much better max function. This is just an example.

```
template <class T>
T& max(T& lhs, T& rhs) {
    return lhs > rhs ? lhs : rhs;
};
```

Pretty trivial, but powerful. Lets see it used in a couple of examples. It should be obvious that this works with numbers:

```
int a=1;
int b=2;
cout << max(a,b) << endl; //prints 2

float c=22.4;
float d=32.456;
cout << max(c,d) << endl; // prints 32.456
```

But this also works with std::strings, because string implements `operator>`;

```
string foo("foo");
string bar("bar");
cout << max(foo,bar) << endl; // should print "bar"
```

It should even work for user defined types. Let's implement a simple Employee class to assure ourselves of this. We will have two private variables - a string for a name, and an int for an employee id. We will implement a constructor, the >operator, and the overloaded stream output operator so that we can cout the employee.:

```
class Employee {
    std::string name;
    int employee_id;
public:
    Employee(const std::string& name_, int id) : name(name_), employee_id(id) {};
    // notice that we consider employees with smaller ids greater than employees with
    // larger ids.
    bool operator>(const Employee& other) { return this->employee_id <= other.employee_id; }
    // getters
    const std::string& getName() const { return name; }
    const int& getId() const { return employee_id; }
    // stream operator to cout class
    friend std::ostream &operator<<(std::ostream &os, const Employee &employee) {
        os << "<" << employee.name << ", id: " << employee.employee_id << ">";
        return os;
    }
};
```

And now lets use it:

```
Employee jg("Jonathan Gerber", 1234);
Employee dr("Doug Roble", 4);

cout << max(jg,dr) << endl; // should print out doug rouble
```

Template Specialization

Sometimes a template won't work for a specific type or class. In this case, you can implement a specific version targeting the offending type or class. Certainly, this shouldn't be your first choice. Say, for instance, that the class in question doesn't supply an appropriate operator. If adding the operator, either as an additional method, or as a free function, doesn't make sense, then you can specialize the template.

The way you go about doing this is simple. First, lead off with `template <>` (notice the lack of types). Then, you declare your actual class or function with a suffix consisting of open angle bracket, type, and close angle bracket (eg

`max<Person>`). Lastly, it is a simple matter of replacing any reference to the existing type variables with concrete types. So, for example lets say we want to use the actual name to determine the result of max based on the name. Here is how we would do that:

```
template <>
const Employee& max<Employee>( Employee& a,  Employee& b) {
    return a.getName() > b.getName() ? a : b;
}
```

After coding up this specialization, you should notice that the max call with Employee will provide a different result, based on alphanumeric sorting of the names.

Variadic Templates

TODO

Meta Programming

I am not going too deep here, but you can have a whole lot of fun with templates. The whole Standard Template Library is written as templates, and much of boost is as well.