CISC 3130
Exercise 0

Introduction to the STL

In this exercise:

- you will be introduced to the C++ Standard Template Library
- you will be introduced to the idea of *generic* code and the techniques for using /writing generic code in C++
- you will compile, execute, and analyze code written using the STL

Introduction

Chapter 18 of the *C++ Annotations* book begins

> The Standard Template Library (STL) is a general purpose library consisting of containers, generic algorithms, iterators, function objects, allocators, adaptors and data structures. The data structures used by the algorithms are abstract in the sense that the algorithms can be used with (practically) any data type.

That is a LOT of nouns. Over the course of the semester, you'll start to get a sense of what most of these things are. Today we'll spend a little time with containers/data structures, iterators, and generic algorithms.

Generic Code

Most of these concepts are part of the general idea of *generic* code. The word "generic" has a few meanings in English—"predictable and unoriginal," "having no brand name," "relating to a group of things; not specific." In some sense, the idea of generic code draws on all of these: in essence, generic code is code that doesn't rely on any particular type of data.

The most fundamental element of C++ that supports generic code is the *template*. You've probably read and/or written template-based code before now; Chapter 21 of *C++ Annotations* has a good treatment and useful guidelines; you may also want to review an introductory C++ textbook's discussion of templates.

Containers

Chapter 12 of C++ Annotations has a detailed summary of the essential containers provided in the STL. As the book says,

> The datatypes discussed in this chapter are all containers: you can put stuff inside them, and you can retrieve the stored information from them.The interesting part is that the kind of data that can be stored inside these containers has been left unspecified at the time the containers were constructed. That's why they are spoken of as abstract containers. Abstract containers rely heavily on templates, covered in chapter 21 and beyond. To use abstract containers, only a minimal grasp of the template concept is required.

Let's start with the array container discussed in 12.4.1 (and summarized in this table).

Compile and run the following short program (copy-and-paste if you like). Make sure you use the C++11 standard when you compile; e.g. `$g++ -std=c++11 exercise0.cpp`

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

int main()
{
    std::array<int, 10> testArrayInt = {10, 9, 8, 1, 2, 3, 6, 5, 4, 7};
    std::array<std::string, 2> testArrayString =
                                          {std::string("dog"), "cat"};

    for(const auto& s: testArrayInt)
       std::cout << s << ' ';

    for(const auto& s:testArrayString)
        std::cout << s << ' ';

    std::cout << std::endl;
}
```

Syntactically, how can you tell there are templates involved? When you "instantiate" the array template, what information do you need to provide?

If you've not seen this kind of `for` loop in C++ before, it's called a "ranged" `for` loop (it's similar to the "enhanced" `for` loop in Java). What do you suppose the word `auto` means?

What if we decide we want to use a container that doesn't have a fixed size? Two options are the `vector` and `list` containers. Change the two declarations to use one or both of these containers (be sure to `#include` the appropriate headers). Does the program compile and run?

Now change the type of the first container (`testArrayInt`) so that it contains `doubles`. Change a couple of the initial values to be `doubles`, too (like `10.5, 9. 8.2, ...`). Does the program compile and run?

Iterators

The ranged `for` loop seems magically to know it's supposed to start at the "beginning" of the array and stop at the "end." If we were using the C++ built-in array type, this might not be surprising—we could imagine the compiler being able to figure that out. But the loop *also* works on these other containers—`array`, `list`, `vector`—even though these things might have very complicated internal structure. What's going on?

The key is the idea of an *iterator*. As the book says in 18.2, "Iterators are objects acting like pointers." Essentially, iterators "point" to the invidual elements contained by a container—and thanks to

operator overloading, we can use the dereference operator `*` to "get our hands" on the element that an iterator is "pointing" to. More importantly, iterators also give us the *next* element, through the `++` operator. Iterators, then, let us "look at" all the elements in a container even if we don't know how a container is organized internally.

The containers in the STL provide several methods that give us access to iterators for their contents. In particular the methods `begin()` and `end()` return iterators that "point" to the first element and the "past-the-end" element. (So we know that we've seen everything in a container if our iterator `== end()`.) And *these* are what the ranged for loop automatically uses in our code.

Iterators have all kinds of uses in the STL. For example, add these two lines before the output (change the name of the "array" variables if you've already done so in your code):

```
sort(testArrayInt.begin(), testArrayInt.end());
sort(testArrayString.begin(), testArrayString.end());
```

What happens when you run the program? What is the default behavior of this `sort()` function?

Add a little snippet of code that uses the `max_element()` function to print the value of a maximum value of one of these containers.

Generic Algorithms

`sort()` and `max_element()` are two examples of the many *generic algorithms* provided by the STL. "Generic" means "doesn't rely on any particular type of data"—so is calling these "generic algorithms" accurate? If so, how do these two functions avoid the need for specific type names? Is there *any* requirement for the element-types in the containers being processed?

Find something in this algorithms library that will make it easy to compute the sum of the elements in testArrayInt. Write that code snippet and make sure it runs.