# COMP322: Assignment 12- Winter 2012

Due at 11:30pm EST, 29 Feb 2012

## 1 Introduction

For this assignment, we will how we can use the C++standard library to store things in an easier way than we did on assignment one. It is important to remember that although you are now able to use functions and libraries that someone else wrote, the libraries are still representing things the same way. Although much of the memory management is now a bit hidden from you, you still need to keep in mind how each of these operations works in order to assure proper use of it.

On this assignment, you will:

- Learn how to read from a file using a stream
- Practice using *iterators*
- Use two container *classes* in the C++library : `vector` (conceptually like an array) and `list` (conceptually like a linked list). This will provide a brief introduction to object oriented programming as we'll have variables that can "do" things.
- Use the *algorithms* that come with the C++library so that you don't need to write everything yourself.

It will be useful to familiarize yourself with the following pages:

1. `http://www.cplusplus.com/reference/stl/vector/` information on the vector class
2. `http://www.cplusplus.com/reference/stl/list/` information on the list class
3. `http://www.cplusplus.com/reference/algorithm/` Information on the algorithms
4. `http://www.cplusplus.com/reference/iostream/fstream/` Information on reading from files

## 2 Background Information

### 2.1 Container Classes

C++comes with a standard library. The standard library is used to deal with scenarios that occur quite frequently in practice, such as storing an array of elements or a linkedlist of elements. There are several *classes* that are defined in the standard library. The most commonly used of these are `list` (for linked lists), `vector` (for arrays), `set` (you can't add the same element twice), `queue` (provides insertion at one side and deletion from the other side), and `stack` (provided insertion at the same side as deletion). All of these are defined to work on *any* data type and are used in similar ways. For example, a function called `push_back` will always insert an element at the back of any of these data structures (when it's defined). The idea is by keeping the function names similar, it's easy to program since one isn't constantly trying to remember the various names.

These libraries all use a concept known as *templating* or *generics*. The idea here is that many functions that operate on containers do not depend on the *kind* of data being stored in the container. For example to insert something at the front of an array, one needs to shift every element of the array over and then set the 0th element to be the new one. This is done regardless of the kind of data being stored in the array. So it makes sense to be able to define a *generic* data type that will operate on any kind of data.

Whenever one uses these sorts of *generic* or templated classes, one specifies the kind of data being stored using a bracket notation. For example, the following would create a vector of int and a list of double.

```
vector<int> foo;
list<double> bar;
```

You can also, in some cases, create space at the time of creation for data. For example, writing `vector<int> foo(10)` would create space in memory immediately to store 10 ints. You can add elements to the vector regardless using `push_back` for example, but since the vector is representing an array, it will be necessary to resize the array. In addition, some of the functions defined in the algorithms library will depend on space already being allocated.

Each of these containers are *classes*. A class in C++defines a blueprint for what an *object* will look like. A class definition is similar to a struct definition except with the addition that it can *do* things. One can do things *on* an object using a dot notation and then calling a method defined on an object of that type. For example, to add an element to a vector, one could write:

```
vector<int> foo;
foo.push_back(3); //inserts element 3 at end
//Now foo is a vector containing 3.
foo.push_back(5);
//now foo is a vector containing 5
```

These functions are called *on* data and differ from anything we've seen before in C++, where one would have to pass the variable foo to the method as a parameter. Notice also that these methods return void but are changing foo anyway.

There are many functions defined on all of these data types. The best way to learn about them is by practicing with them and by reading the links above.

## 2.2 Iterators

An *iterator* is basically a pointer on steroids. An iterator can be used to *iterate* over every element in a collection. The actual iteration is done differently depending on what the collection is. For example, to iterate over an array, you could create a counter variable *i* and continue to add 1 to it until you reached the end of your array (denoted by the fact that i was equal to the array's length). To iterate over a linked list, you could create a pointer `p` and continue to set `p = p->next` until which point p is NULL.

An iterator is a pointer to an element, that in addition to pointing to a particular element, also has the ability to "jump" to other elements. In the case of *forward iterators* this can be done using the `++` operator, which is very convenient as one can *abstract* the notion of moving forward. One does not need to remember whether it is `i = i + 1` (array) or `p = p->next` (linked list) or a 3rd entirely different thing.

There are several categories of iterators (not mutually exclusive):

- Forward iterators (to go from front to back once)
- Bidirectional iterators (you can go both forward and backwards)
- Random access iterators (you can move multiple elements at a time)
- Read only iterators (only let you read data)
- Write only iterator (data is not defined but you can write into it)

The difference between which kind of iterator you have affects what you can do with it. For example, a bidirectional iterator has the `--` operator defined on it to go backwards. `http://www.cs.brown.edu/~jak/proglang/cpp/stltut/tut.html` and `http://stdcxx.apache.org/doc/stdlibug/2-2.html` are pretty good tutorials on some of the differences.

Generally, for this assignment, we'll be working with forward iterators, both read only and write able.

To use an iterator, you always need to know the *beginning* and the *end*. The end normally refers to an address one past the end. The reason for this is it allows computations such as `end - beginning` to get the size of something without worrying about off by one errors. (This same idea is presumably why it's common to write `for (int i = 0; i < array.length; i++)` instead of `for (int i = 0; i <= array.length - 1; i++)`.)

All of the standard library containers have iterators defined on them. Depending on what kind of container it is, you'll have a different kind of iterator. For example a `vector` has a random access iterator since one can go from any element to another in an array easily. On the other hand a `list` only has a forward or reverse iterator defined on it as it is not possible to go from element X to X+2 for example, without going through X+1. (This is why it's important to understand how a linked list works in the first place or else these explanations wouldn't make sense.)

What this means, is if you have an iterator from a `vector` you can add an arbitrary value to it. For a `list` you'd have to do `++` several times.

You can get an iterators start and end on any of the containers by calling the `begin()` and `end()` methods on a container. Note that `end()` always refers to an element one past the end again. The type of this call will depend on the type of container you have and the type of data you have. For example, if you have a `vector<int>` then the type will be a random access iterator of a `vector<int>`. If you have a `const list<double>` then you will have a read-only forward only iterator of a `list<double>`.

So to iterate over every element, one can write:

```
vector<int> foo;
//add some things to foo
....
//vector<int>::iterator is the type
//if it was a const iterator, you would write
//vector<int>::const_iterator instead
for (vector<int>::iterator current = foo.begin(); current != foo.end(); current++)
{
//do something to the element current
}
```

Remember that an iterator is always essentially a pointer to a specific element. This means that you can dereference it using the * operator. So if I wanted to print every element, then inside the above for loop, I could write:

```
std::cout << "Value is " << (*current) << std::endl;
```

current is a pointer to a specific element, so if that specific element were more complex than a simple type, we could use the dot operator the same way we would with a struct/class. Thus we can also use the `->` operator on iterators over more complex types.

## 2.3 Algorithms

There are many *algorithms* defined that come with the standard library and are included in the header file `algorithm`. Typically these algorithms are very compatible with the container classes and iterators. They can be used to do things such as:

- Given an iterator to a container (both start and finish), count how many elements in the container satisfy a condition.
- Sort elements given a random access iterators start and finish
- Given an iterator's start and finish, and a 2nd iterator's start only, copy elements from the first iterator to the second. Note that in this case, as in most of the algorithm functions, the 2nd iterator is assumed to already have the space needed to fit this. (It's the calling functions responsibility. The reason for this is the writers of the stl algorithms have no idea what you were hoping to do and can't be bothered to guess. So they make it your responsibility.)

Feel free to use any or none of the algorithms on this assignment. (Although at least using the `sort` function seems to be a good idea)

### 2.4 Streams and file IO

The last thing needed for this assignment is a bit of background on streams. Up until now, we have used the `cout <<` structure in a similar fashion to how one uses the `printf()` function in C or the `System.out.println()` method in Java. However, despite appearing to be the same, `cout` is actually very different.

`cout` itself is actually an identifier in C++. It's type is an `ostream` (for output stream). A stream is essentially a *channel* from one place to another. For example, there exists a stream that goes from the program you run to *standard output* which is by default directed to the screen. A stream will typically flow in one direction and if you put information in a stream upstream, it will float downstream. For example, if we send information to `cout` it will flow to standard output, which then flows to the screen.

There is a second stream which flows from *standard input* TO your program. Standard input is typically coming from the keyboard, although it can be linked in other ways as well. `cin` is an identifier that can be used to access this stream. The type of `cin` is `istream`

In C++, streams are handled using the operators `>>` and `<<` . Each of these operators take as input two operands: the stream and the data:

1. istream >> l-value
2. ostream << data

In the first case, we have `istream >> l-value` which means direct the information from `istream` into l-value (i.e. a variable).

In the second case we have direct data from *data* to the `ostream`.

So for example, when we write `cout << "Hello world"` we are actually directing to the `cout` stream the data `Hello world` . If we have a variable `x` and we write `cin >> x` we are directing whatever is currently in the stream `cin` to the variable `x`.

Both of these operators are designed cleverly to *return* the stream on the left side as well. This allows us to put multiple statements together. For example:

```
cout << "hello " << x;
```

is really `(cout << "hello") << x` which means

- Send to the stream `cout` the contents `hello`
- Evaluate the expression `cout << "hello"` (which will have the type of `ostream` and evaluate to be `cout` as well) and send to it the data stored in x

To read from a file in C++, you can do the exact same thing as reading from the keyboard, except you must open a stream to a file. In the case of reading from a keyboard, the stream is already created for you. To do this, you should use a data type called `ifstream` which is defined in the header `fstream`. This object has the `>>` operator defined on it in the same way as `cin` does.

To read from a file, you'll need to use the `open` method, defined on an fstream. You'll also find useful the function `fail()` which returns a boolean of true or false depending on whether the read was successful. This can be used to detect whether you've reached one past the end of the file. Finally, the `close` method should be used when you are done with the stream to clean up after yourself.

All of these methods are defined *on* and object of type `fstream`, so you should use them as one would a container class method. See `http://www.cplusplus.com/reference/iostream/fstream/` and `http://www.cplusplus.com/doc/tutorial/files/` for examples of using these.

## 2.5 Using Templated Iterators

In some cases, we may want to have a function take an iterator as input. However, we want to make our function as general as possible and thus would like to write a function that could take as input any kind of iterator: for example an iterator of a vector or an iterator of a list. In this case, you will need to use the idea of a templated function. To do this, write `template<class InputIterator>` before the function header you are defining.

This tells the compiler that in the subsequent function, if you write `InputIterator` it can refer to different types. You can then use `InputIterator` in place of the actual type. For example, one could make their function header `foo(InputIterator start, InputIterator finish)` and then write commands like `InputIterator current = start;` or `current++` the same way.

The determination of whether the various operations you do with the iterator are valid is made at compile time. (See appendix for a bit on this)

# 3 Practice questions (not for credit)

In order to get a bit of practice with using the standard library and iterators, I suggest writing a C++program to do the following (the idea is since this is not for credit Dan and the TAs will be happy to discuss all solutions to them).

1. Define a struct called `Point2d` that has 2 data types, both ints.
2. Create a `list<Point2d>` and, using the `push_back` function, add some coordinates to it.
3. Create a `vector<Point2d>` and do the same thing.
4. Write a function `printList` which takes as input a `list<Point2d>& numbers` and, *using an iterator* prints the contents of the entire list. To do this, you'll need to set up a loop that starts out at `list.begin()` and continues until the iterator is equal to `list.end()`

   Hint: The for loop header will look like:

   ```
   for (list<Point2d>::iterator number = numbers.begin(); number != numbers.end();
   number++)
   ```
5. Now write the same function except have it take as input a `vector<Point2d> &numbers`

6. Write a 3rd and 4th function that take as input the same as before except they should be *const* vectors.

   Note that since your function takes as input a `const list<Point2d>`, if you call the `begin` method on the `list<Point2d>` you'll get an expression of type `list<Point2d>::const_iterator` . The reason for this requirement in C++is if you had a non read-only iterator, the compiler would not notice a problem if you later on tried to change the data using the iterator. To deal with this, when you declare the variable in the for loop header, you should write `list<Point2d>::const_iterator` instead of `list<Point2d>::iterator`

7. Now, you may be thinking: All of these functions are identical! Isn't there a better way to combine these functions. And you'd be right. This is where iterators come in handy. If we make our function take as input an *iterator* of a container instead of the container itself, we'll be able to use the same function. The only question is what to make the function header. To do this in C++, you'll need to take advantage of *templates* We'll talk more about templates later on, but for now, before every function like this, you should add the line:

   ```
   template<class InputIterator>
   ```

   This tells the compiler that in the subsequent function, when you use the identifier `Iterator`, it will be in place of another type that is determined *at compile time* based on how the function is called.

   You could then make your header take as input `InputIterator start` and `InputIterator finish` and your for loop would be similar—it would now declare a variable of type `InputIterator`. Note a few things:

   - `InputIterator` is just the term we'll use to identify the TBD type. It could have been anything we wanted.
   - You'll have to write the template declaration before every function you want to do something similar in.
   - The compiler will check at compile time if you are using this call correctly. For example, if you access an elements `x` coordinate, then the function can't be called on a container that has a datatype without an `x` (e.g. you could call it on a `list<Point2d>` but not a `list<int>`
   - The compiler will also check at compile time if you are passing the *right* kind of iterator. For example, if you use the iterator to *change* the contents and you pass it an iterator to a const container, you'll get a compiler error.
   - The variable `finish` normally will be one PAST the end of the collection. The standard library was designed to have all of the containers `end()` methods give you an element one past the end. The motivation of this was so that one could make lines like `int size = collection.end() - collection.begin()` without having to worry about off by one errors.

## 4 Assignment Requirements

For this, you should write the following functions and data types. You may wish to add a `main` method to test the code, but you should comment this method out or delete it entirely before handing in your code.

- Define a struct `RatingInformation` which has three properties, all `int` representing the userId, the movieId, and the rating.

- Define a function `readFromFile` that takes as input a `char* file` and returns a `list<RatingInformation>` (you need to write `#include<list>` to use this type `list` from the standard library. This method should take as input a file which contains triplets (separated by a space or tab) of userId, movidId, and ratings. For example, if the file looked like:

  ```
  3 4 5
  10 4 9
  ```

  it would be read as "User 3 has given movie number 4 a rating of 5. User 10 has given movie number 4 a rating of 9" You may assume that the file is well formatted in that there is a perfect multiple of 3 and all elements are integers.

  Your method should parse the file using an `ifstream` object and fill in a list of rating informations. This list should them be returned.

  Note that to call this method, you can pass a string literal, that will then be converted to a char*. For example, one could write `readFromFile("movieratings.txt")`

- Write a function `print` which using the template approach described in the practice problem, takes as input a `InputIterator start` and an `InputIterator end` and prints the contents of the entire rating collection. For example, if the rating collection is what appeared in the file below, your function should print something like:

  ```
  UserId is 3, MovieId is 4, and Rating is 5
  UserId is 10, MovieId is 4, and Rating is 9
  ```

  **Note that the variable `end` will refer to one PAST the end of the collection. So you should NOT test on this spot.**

- Write a function `transform`. This method should return `void` and take as input a `vector<RatingInformation>& user` and an `int shift`. It should iterate over every element in the `vector` and add to each rating the shift. For example, if the ratings started out as $10, 3, 4$ and the shift is $-6$ then the ratings should end up $4, -3, -2$

  You may choose to do this using a "conventional" for loop with a variable i or an approach using iterators. I recommend using the iterator approach as it's more consistent with the other required functions. Note that since your function takes as input a *reference* to a vector, if you change the vector elements in the `transform`, it will affect the data in the calling function as well.

- Write a function `dotProduct` which takes as input two `const vector<RatingInformation>&` Your method may assume that the vectors are the same size and that the `RatingInformations` inside the vectors are both sorted by movie id, and they contain the exact same list of movies. Your method should then take the dot product of the *ratings* of the corresponding vectors. This will always be an int since all ratings are int.

- Write a function `containsMovie` which returns a bool. Your method should take as input three things, an `InputIterator start`, an `InputIterator finish`, and an `int movieId`. It should return `true` if any of the items between start and finish have the same movieId as the int passed in as input. It should return `false` otherwise. Remember that to do this, you will need to declare the template before the function again.

- Write a function `getSubset` which takes as input 4 things: `InputIterator user1Start, InputIterator user1Finish, InputIterator user2Start, InputIterator user2Finish`. It should return a `vector<RatingInformation>` consisting of all of the `RatingInformations` that are in user1's collection for which there is a corresponding rating in user2's collection.

  That is, it should iterate over each element from `user1Start` until `user1Finish` and check to see if there is a movie with the same movie id between `user2Start` and `user2Finish`. If it is part of this, it should add the element to the `vector<RatingInformation>`. Otherwise it should not.

  Of course, you may use the function `containsMovie` defined above.

- Finally, write a function `computeSimilarity` which will use all of the above functions to calculate the similarity between two users. This is an important step in a movie recommender system because we can make recommendations based on the idea that similar users like similar movies.

  Your function should take as input a `list<RatingInformation> ratings`, and two ints, representing the ids of user1 and user2. Your method should then do the following to calculate the similarity and return an `int`.

  – First, put all of the `RatingInformations` belonging to user1 into one of the C++containers. There are several ways to do this. One option would be to write a function to return a `list<RatingInformation>` and, using the `push_back` function and an iterator, add appropriate values. A second, perhaps faster way, would be to use the function `remove_copy_if` which allows you to, given an existing collection, copy elements to a new collection that DONT satisfy a constraint. Be careful though to use this function correctly. It may take a bit more time to learn the first time, but once you learn how to use it correctly you'll be good to go forever :) [1]

[1] You have to be careful that the new collection you use has enough space allocated for it, meaning you may need to use the `count_if` function as well to determine this ahead of time. Both of these functions are defined in the `algorithm` library (meaning you should include it). An example of how to use either of these is at `http://www.cplusplus.com/reference/algorithm/remove_copy_if/` Note that since we have not yet

- Next do the same thing with user2.
- Now use the `getSubset` function you wrote to get the overlapping list of `RatingInformation` for both users.
- Transform both of these using the transform function you wrote. You should use a shift of negative five.
- Now *sort* both lists of ratings according to their movie id. To do this, you should call the `sort` function that is part of algorithms. The `sort` function takes as input 3 things. A start iterator, an end iterator (one past the end of the collection as usual) and a *function pointer*. The function pointer is used to denote what *less than* means in this context. It will then sort them accordingly. This means you should define a function that takes as input two `RatingInformation` and returns whether the first one is less than the second one or not. For example, if you wrote a function with header `bool compareMovieIds(RatingInformation r1, RatingInformation r2` you should return whether r1's movie id is less than r2's movie id. Then if you wanted to sort a `vector<RatingInformation> ratings` you could do so by writing `sort(ratings.begin(), ratings.end(), compareMovieIds)`. An example of this is at `http://www.cplusplus.com/reference/algorithm/sort/`
- Finally, return the dot product of the 2 sorted vectors.

## 5 Using/installing g++

**g++** is the GNU C++ compiler. A recent version (4.3.2) is available on most of the Ubuntu lab computers. Our solution was developed using version 4.4.1.

We build our solution using the command line:

```
g++ -Wall -o MovieRecommender.exe MovieRecommender.cc
```

If you want to install **g++** on your system, you should be able to find instructions around the web.

For the Mac, the instructions here look reasonably accurate and up-to-date:
`http://www.edparrish.com/common/macgpp.php`

For PC/Windows systems, there are multiple choices, but the Cygwin environment mimics most of the Linux command line on Windows, so it is a good option:
`http://www.cygwin.com`

For Ubuntu (and possibly Debian) users, you can install the free `g++` package with the following command:

```
sudo apt-get install g++
```

If you have trouble, or you're running some other operating system, let us know and we'll try to help you.

covered classes, if you choose to use this approach, you will perhaps need to use a global variable to store the userId to compare things with. (You are free to define a class which can take a constructor as input if you like as well.)

## 6 Submitting your assignment

### What To Submit

`MovieRecommender.h` Here you should put all function headers.
`MovieRecommender.cc` Here you should put all code. You should include the header file. Remember to delete your main function before handing the assignment in (or comment it out)
`Confession.txt` (optional) In this file, you can tell the TA about any issues you ran into doing this assignment. If you point out an error that you know occurs in your problem, it may lead the TA to give you more partial credit. On the other hand, it also may lead the TA to notice something that otherwise he or she would not.

## 7 Aside: a little philosophy

Both Java and C++come with a standard library with many very useful functions/methods/classes defined in them. Both of them, for example, have classes defined that can be used to represent arrays (vector in C++and ArrayList in Java) and linked list (list in C++and LinkedList in Java). They both try to make these libraries compatible with each other so that one can write the same code that operates regardless of if they have an arraylist/vector or a linkedlist/list (when possible of course. In some cases the problem doesn't allow it.) Both languages allow you to design code to work on many different kinds of collections. However, they do it differently: In Java, they use the notion of interfaces, in C++, they use the notion of iterators.

In Java, they define a notion of a *Collection*. A `Collection` is an *interface* which has several methods defined on it such as `add(), size(), isEmpty(), iterator()`—things that any collection, no matter what kind of collection, should be able to do. ArrayLists and LinkedLists then *are* Collections. So anywhere that a Collection is expected, it is OK to provide an `ArrayList` or `LinkedList` since they are indeed a `Collection`. The significance of Collection being an interface is that you can't create an instance of Collection without creating an instance of some other subtype of it.

This means, that if I wanted to define my method to work on both `LinkedList` and `ArrayList`, I should actually define my method to work on `Collection`. Then, at run-time, the Java engine will determine which one is actually there. Both LinkedList and ArrayList classes have in common, in their definition, that they come from the same parent interface.

In C++there is no direct concept of an interface (although there are some similar things one can do using inheritance). There is nothing intrinsically linking a `list` data type to a `vector` data type. Instead, the way to make your functions general is to define them to take as input an *iterator* to the data. Since all the containers have these iterators, the algorithm functions work well on these.

The significance of all this comes when you are writing methods to try to make them compatible with standard library functions/methods. In Java, if you defined your own class to hold a list of elements, you should make sure it implements the interface Collection. Then code using Collection will automatically work with your new class. Note that doing so in

Java will force you to define an iterator there as well (by writing a `next()` and `hasNext()` method–with this you can use the foreach syntax for a for loop). If you choose to write an algorithm in Java, and want it to work on many kinds of collections, you should write your method to take as input a Collection instead of the more specific ArrayList.

In C++, when you define your class, you would want to define a `.begin()` method and a `.end()` method. This will make your class compatible with the standard library algorithms. If you are trying to write an algorithm to be compatible with the C++containers, you should similarly have them take as input iterators. You must make these iterators using the template idea, since they can be iterators of any kind. This template is then assigned a type *at compile time* which is different than Java's which occurs at run time.

This is a bit different philosophy between the two languages, but the common goal is to try to reduce the amount of code one has to write by making things as general as possible. Java's approach is a little more straightforward in that one can iterate over the arrays without really knowing it. OTOH C++'s approach is more general. For example, to iterate over an array in *reverse* one simply has to define an iterator in which the beginning is at the *end* of the array and then define the `++` operator to decrease the address in memory being looked at. Indeed, this is done in the `vector` class using the `.rbegin()` and `.rend()` methods. In Java one would have to create an entirely new collection to do this sort of thing or actually change the order of the elements.