Review from Monday’s Class
Koenig & Moo: Chapter 4
- Using functions to improve clarity and promote code re-use
- Separating a program across multiple files.
- Structs for organizing data.
- `sort; sort` on `struct` objects; the `less_names` function.

Today’s Class
Koenig & Moo: Sections 5.1-5.5
- Program to maintain class list and waiting list
- Erasing items from vectors is inefficient
- Iterators and iterator operations
- Lists as a different sequential container class.

Example: Course Enrollment and Waiting List
Program in `classlist_vec.cpp` to build the class list and the waiting list for a single course.
- The program is structured to handle interactive input.
- Vectors store the enrolled students and the waiting students.
- Error checking ensures the input is valid.
- The main work is done in the two functions `enroll_student` and `remove_student`.
- The invariant on the loop in the main function determines how these functions must behave.
• The vector member function `erase` removes the indicated value from the vector. The particular example we will discuss in class is

\[
\text{enrolled.} \text{erase( enrolled.} \text{begin()} + \text{loc } );
\]

**Inefficiency of Erase from Vectors**

• This follows from the need to be able to index the vector. To see, this, here's what the erase function actually does:
  
  – The item indicated by the iterator is removed.
  – All items “above” the iterator (corresponding to a higher index), are copied down.
  – The size is reduced by one.

• This is very expensive when the vector is large and the item to be erased is near the beginning of the vectors.

**What To Do About It?**

• We need a different sequential container, called a *list.*
  
  – This has a “linked” structure that makes the cost of erasing independent of the size.

• We will move toward a list-based implementation of the program in two steps:
  
  – Rewriting our `classlist_vec.cpp` code in terms of *iterator* operations.
  
  – Replacing vectors with lists

**Iterators**

Definition (from the text):

• Identifies a container and a specific element stored in the container.

• Lets us examine (and change, except for const iterators) the value stored at that element of the container.

• Provides operations for moving (the iterators) between elements in the container.
• Restricts the available operations in ways that correspond to what the
container can handle efficiently.

As we will see, iterators for different container classes have many operations
in common. This often makes the switch back-and-forth between containers
fairly straightforward.

**Iterator Declarations and Operations**

- Iterator types are declared by the container class. For example,

```cpp
vector<string>::iterator p;
vector<string>::const_iterator q;
```

defines two (uninitialized) iterator variables.

- The dereference operator is used to access the value stored at an ele-
ment of the container. The code

```cpp
p = enrolled.begin();
*p = "012312";
```

changes the first entry in the `enrolled` vector.

- The dereference operator is combined with dot operator for accessing
structs stored in containers. For example,

```cpp
vector<StudentRec>::iterator i = students.begin();
(*i).first_name = "Samantha";
```

Notes:

- This operation would be illegal for a `const_iterator`.
- The parentheses on the `*i` are required!!

- There is a “syntactic sugar” for the combination of the dereference
operator and the dot operator, which is exactly equivalent but simpler:

```cpp
vector<StudentRec>::iterator i = students.begin();
i->first_name = "Samantha";
```

- Iterators can be incremented and decremented using the `++` and `--`
operators to move to the next or previous element of any container.
• Iterators can be compared using the == and != operators.
• Iterators can be assigned, just like any other variable.
• Vector iterators have several additional operations:
  – Integer values may be added to them or subtracted from them. This leads to statements like
    \[
    \text{enrolled}.\text{erase}( \text{enrolled}.\text{begin()} + \text{loc} );
    \]
    which we have already seen.
  – They may be compared using operators like <, <=, etc.
  – For most containers, these “random access” operations are not legal and therefore prevented by the compiler. Reasons will gradually become clear.

Revising the Class List Program to Use Iterators
During lecture, we will revise the class list program to use iterators. The result will be posted on the web.

Lists
• Our second standard-library container class
• Lists are formed as a sequentially linked structure instead of the array-like, random-access / indexing structure of vectors.
  – Pictures showing the differences will be drawn in class.
• Lists have push\_front and pop\_front functions in addition to the push\_back and pop\_back functions of vectors
• Erase is very efficient for a list, independent of the size of the list.
• We can’t use the standard sort function; we must use a special sort function defined by the list type.
• Lists have no subscripting operation.

Revising the Class List Program to Use Lists and Iterators
In class we will rewrite this program again to use lists and list iterators. The resulting program will be posted on the web.
Looking (Way) Ahead

- Although the interface (functions called) of lists and vectors and their iterators are quite similar, their implementations are VERY different.

- Clues to these differences can be seen in the operations that are NOT in common, such as:
  - No indexing (subscripting) in lists.
  - No `push_front` or `pop_front` operations for vectors.
  - Several operations invalidate the values of vector iterators, but not list iterators:
    * `erase` invalidates all iterators after the point of erasure in vectors;
    * `push_back` invalidates ALL iterators in a vector
    The value of any associated vector iterator must be re-assigned / re-initialized after either operation.
  - Lists have their own, specialized `sort` functions.

- Later in the semester we will see how to implement lists and vectors. This will explain their behavior and also introduce “lower-level” C++ operations.