Chapter 20
Containers and Iterators

Examples of common tasks we do with lots of data

- Find a number in a phone book, given a name
- Find the highest temperature
- Find all values larger than 8800
- Find the first occurrence of the value 17
- Sort words in dictionary order
- Sort temperatures in descending order
- What is the largest temperature seen?
- What are the top 10 best-sellers?
- What is the sum of the elements?

Characteristics of these tasks

- We can describe them each without actually mentioning how the data is stored
- We must be dealing with lists, vectors, files etc., for these tasks to make sense but we do not have to know the details about how the data is stored to talk about what to do with it
- What is important is the type of the values, how we access them and what we want to do with them

Abstract view of what we do with data

- Collect data into containers
  - Such as vector or list
- Organize data
  - For printing
  - For fast access
- Retrieve data items
  - By index (e.g., get the **N**th element)
  - By value (e.g., get the first element with the value **"Chocolate"**)
  - By properties (e.g., get the first elements where "**age<64**")
- Modify a container
  - Add data
  - Remove data
  - Sort data
- Perform simple numeric operations

Ideals

We would like to provide code for common programming tasks so that we do not have to re-do the work each time we find a new way of storing the data or a slightly different way of interpreting the data

- Finding a value in a **vector** isn't all that different from finding a value in a **list** or an array
- Looking for a **string** ignoring case isn't all that different from looking at a **string** not ignoring case
- Graphing experimental data with exact values isn't all that different from graphing data with rounded values
- Copying a file isn't all that different from copying a vector

Pseudo code for computing maximum value

```
max = first element
current_element = second element
while current_element is within the group of elements
   if current_element > max, then
   max = current_element
advance to the next element
end while
```

Let's implement this for lists

```
struct Element
{       int value;
        struct Element * next; // singly-linked list
};
// assume that main() has initialized a list with some elements
int max = list->value; // list points to the first element
struct Element* current = list->next; // points to second element
while (current != NULL) // test if within the group of elements
{
   if (current->value > max)
   {
      max = current->value;
   }
   current = current->next; // advance to next element
}
```

Let's implement this for arrays

```
// assume that main() has initialized a array with some elements
int max = *array; // first element of the array, use of * is intentional
int* one_past_end = array + size;
int* current = array + 1;   // starts at second element
while (current != one_past_end) // test if within the group of elements
{
  if (*current > max)
  {
    max = *current;
  }
  current++;     // Advance to the next element
}
```

Surprise!

- Both pieces of code are almost identical
- I used pointer notation for the array to make them look even more similar
- In both cases:
  - We have a pointer pointing to the current element
  - This pointer is compared to a particular value to test if we are within the group of values
  - The pointer is dereferenced to obtain the particular value
  - This pointer allows us to advance to the next element

C++ STL

- STL: Standard Template Library
- Has **containers and iterators**
- Container: an object that represents a group of elements of a certain type, stored in a way that depends on the type of container (i.e., array, linked list, etc.)
- Iterators: pointer-like object (that is, an object that supports pointer operations) that is able to "point" to a specific element in the container

STL containers

- Examples: list, vector, set, multiset, map
- Containers support operations, such as:
  - insert
  - erase
  - push_back
  - push_front
  - pop_front
  - pop_back

STL iterators

- STL takes advantage of the operator overloading feature C++ to provide class definitions for the iterators to represent pointer-like objects
- Basic operations required for pointers are:
  - assigning a value (to make it point to a specific data item)
  - dereferencing it (to access the data element - in read or write mode)
  - pointer arithmetic (in particular, incrementing or decrementing the pointer with the ++ or -- unary operators), and comparison of the values (in particular, for equality or inequality)

Operations supported on iterators

- It points to an element of a sequence
- You can compare two iterators using == and !=
- You can refer to the value of the element pointed to by an iterator using the unary * operator
- You can get an iterator to the next element using ++

Implementing containers and iterators

- I showed an example “List” container in the last lecture: see listcontainer.cpp under course resources
- The example also defined the “iterator” class for this container
- Today, we will see example usages of STL iterators and containers