The Standard Library and Generic Algorithms

PIC 10A, UCLA
©Michael Lindstrom, 2015-2019

The Standard Library and Generic Algorithms

Here we will look into just a few other useful data structures besides **std::vector**s and **std::string**s, as different applications may require different structures.

In addition, we will look into some very powerful **generic algorithms**: procedures that can be applied across many different containers and allow programmers to make use of robust and efficient methods to manipulate data.

std::vector and std::string

A **std::vector** (and **std::string**) stores data **contiguously**.

This contiguous storage allows for fast access to individual elements with the subscript operator **[]**.

We can only **insert or remove** data efficiently on one end with **push_back** and **pop_back**, but inserting an element at an arbitrary position can be costly.

These contiguous containers have iterators that obey pointer arithmetic, such as

```
it += 3; // advance 3 positions forward
```

std::set

A **std::set** is a templated class found in the **<set>** header that behaves as a mathematical set: it stores elements of a given type and **elements are stored uniquely** (more or less...), so a given value can only appear once.

For example in mathematics, we have $\{1, 2, 4\} \cup \{2\} \cup \{6\} = \{1, 2, 4, 6\}$ where $\cup$ denotes a union. Note the **2** only appears once in the overall result.

Elements can be **quickly inserted into and erased from** a set.

std::set

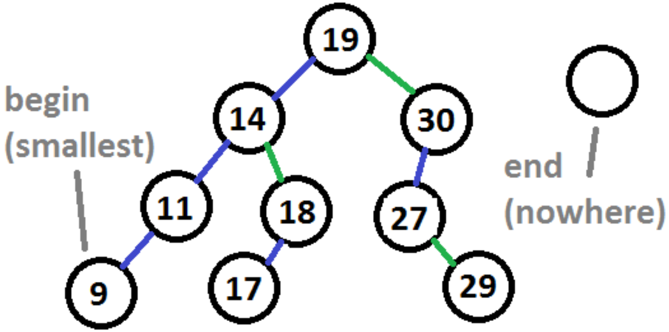
The data in a set are stored with an ordering based on **operator<** by default.

The **begin** function returns an iterator to the "smallest" element and the **end** returns an iterator past the "largest" element, i.e. pointing nowhere.

The data in a set **are not stored contiguously**; as a result, we cannot go to an arbitrary position in the set directly. We have to start at the beginning or end, and move our way through...

std::set

An abstract representation of how data in a set could be represented.



There is order: green and down, values go up; blue and down, values go down.

std::set

While set iterators can be incremented and decremented

```
++it; // go to next element
- -it; // go to previous element
```

they do not observe pointer arithmetic

```
it += 1; // ERROR: cannot add/subtract offset to iterator
it = it -7; // ERROR
```

std::set

We will look at a few member functions of a **std::set**.

- Default constructor, for example

  ```
  set<int> mySet; // empty set to store ints
  ```

- Insertion operator: **insert** inserts values into the set

  ```
  mySet.insert(3);
  mySet.insert(7);
  mySet.insert(11);
  mySet.insert(-8);
  mySet.insert(3); // will not insert 3 again - elements are unique!
  mySet.insert(-2);
  mySet.insert(0);
  ```

- Compute size - of type **size_t** - with **size**:

  ```
  mySet.size(); // == 6
  ```

std::set

- Member functions **begin** and **end**

```
// mySet stores -8, -2, 0, 3, 7, 11

auto beg = mySet.begin(); // beg points to -8

auto end = mySet.end(); // end points "past" the 11
- -end; // end points to the 11
cout << *end << endl;
```

```
11
```

- Erase an element if the element exists (or do nothing if it is not there) using **erase**

```
mySet.erase(0); // removed 0 element
mySet.erase(111111); // does nothing
```

std::set

- Use **find** to find an element in the set: returns an iterator to the given element if found, otherwise returns the past-the-end iterator.

```
// mySet stores -8, -2, 3, 7, 11

auto it1 = mySet.find(-2); // it1 points to the -2
auto it2 = mySet.find(125); // it2 points to end

if( it1 != mySet.end() ) { // if element was found
    cout << "Found value "<< *it1 << endl;
}

if ( it2 == mySet.end() ) { // if failed to find
    cout << "Failed to find value.";
}
```

```
Found value -2
Failed to find value.
```

std::set

- Loops can be done with iterators, for example:

```
// for all positions
for( auto iter = mySet.begin(); iter != mySet.end(); ++iter) {
   cout << *iter <<" "; // print value at the position with space
}
```

  or via range-for:

```
for ( int i : mySet ) { // for each int in set
   cout << i << " "; // print it with a space
}
```

  Both produce:

```
-8 -2 3 7 11
```

std::list

A **std::list** is a templated class found in the **<list>** header.

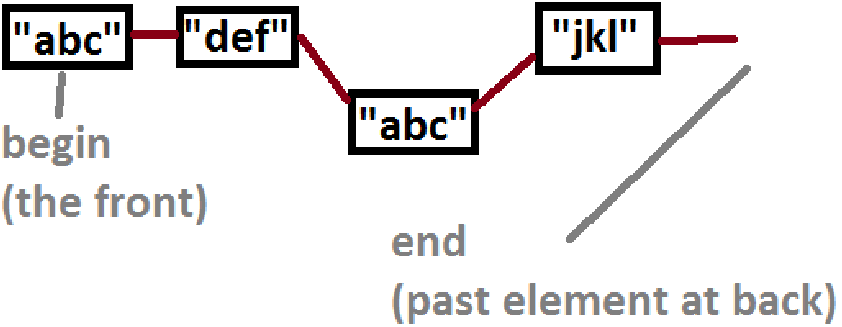
Elements can be **quickly inserted/removed at the front and back**.

Given a specific position, given by an iterator, insertion/removal at that position can also be very quick.

As with a **std::set**, the data are not stored contiguously; also like a **std::set**, the iterators do not obey pointer arithmetic.

std::list

Diagram of how data in a list could be stored.



Elements need not be stored uniquely, nor in sorted order. The data are not stored contiguously.

std::list

We consider a few functionalities.

- Default constructor, for example

  ```
  list<string> myList; // empty list to store strings
  ```

- Insert at back or front with **push_back** and **push_front**

  ```
  myList.push_back("are"); // == { "are"}
  myList.push_front("how"); // == { "how", "are"}
  myList.push_front("Hello"); // == { "Hello", "how", "are"}
  myList.push_back("you"); // == { "Hello", "how", "are", "you"}
  myList.push_back("..."); // == { "Hello", "how", "are", "you", "..."}
  ```

std::list

- Remove from front or back with **pop_front** and **pop_back**

```
myList.pop_back(); // == { "Hello", "how", "are", "you"}
myList.pop_front(); // == { "how", "are", "you"}
```

- Compute the size with **size**:

```
myList.size(); // == 3
```

- Member functions **begin** and **end**

```
auto beg = myList.begin(); // points to "Hello"
auto end = myList.end(); // points past "you"
```

std::list

- Insert a value before a given iterator position with **insert.** For example, inserting before end, adds an element to the end.

  ```
  myList.insert(end, "today"); // == { "how", "are", "you", "today"}
  ```

- Erase an element from a given position with **erase**

  ```
  end = myList.end(); // points past the end
  - -end; // points to "today"
  myList.erase(end); // == { "how", "are", "you"}
  ```

std::list

- Range-for loops can be used (along with loops with iterators).

```
// myList == { "how", "are", "you"}

for(const auto& s : myList ) {
   cout << s << " ";
}
```

```
how are you
```

std::pair

A **std::pair**, found in **<utility>**, is a templated class that can store two (possibly different) value types and has public members **first** and **second.**

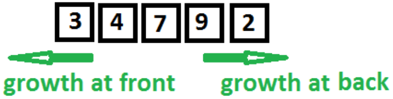
```
// stores a paring of a string and double
pair<string, double> studentGrade;

studentGrade.first = "Joe Bruin";
studentGrade.second = 93.8;

cout <<studentGrade.first << ", " << studentGrade.second << "%";
```

```
Joe Bruin, 93.8%
```

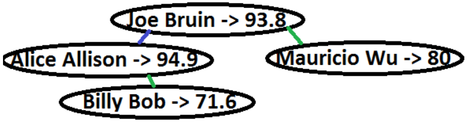
Other Data Structures

There are many more containers. Just to name a few...

- **std::deque** like a **std::vector** with subscripting [], etc., but it can grow from both ends. Data are not contiguous - can be useful for very large amounts of data.



- **std::map** effectively behaves as a set of key-value pairs. Each key has a unique value ascribed to it. Entries are sorted by key.



Other Data Structures

- **std::multiset** is a set that can store multiple instances of a value.
- **std::multimap** is a map where a key can have multiple values.
- Hash maps: these are specialized data structures to handle **vast amounts of data**, while preserving **rapid insertion, removal, and lookup**, regardless of how many data are stored.

  The C++ standard does have containers that are implemented as hash maps, too.

Different Types of Iterators

Most containers can have 4 different types of iterators, depending on which direction they move and whether they "promise" not to modify the elements of the container.

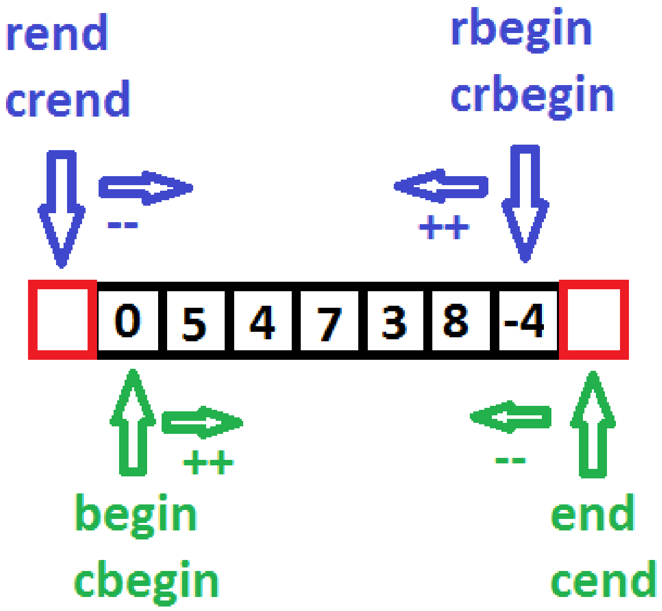
- **iterator**s can access/modify elements and they move "forwards". These are returned by **begin** and **end** functions.
- **const_iterator**s can access elements and they move "forwards". These are returned by **cbegin** and **cend** functions.
- **reverse_iterator**s can access/modify elements and they move "backwards". These are returned by **rbegin** and **rend** functions.
- **const_reverse_iterator**s can access elements and they move "backwards". These are returned by **crbegin** and **crend** functions.

Different Types of Iterators

For **forward iterators**, **begin** and **cbegin** point to the first element and **end** and **cend** point past last element! **++** moves towards the end, **–** moves towards the beginning**.**

For **reverse iterators**, **rbegin** and **crbegin** point to the last element and **rend** and **crend** point before the first element! **++** moves towards the beginning, **–** moves towards the end**.**

Different Types of Iterators



Different Types of Iterators

While the container classes of the C++ Standard have **begin** and **end** member functions, a C-style array does not.

To add more versatility to our code, we may use the functions

```
begin(container);
crend(container);
// etc.
```

instead of invoking container-specific member functions

```
container.begin();
container.crend();
// etc.
```

because the **begin**, **crend**, etc. non-member functions of **<iterator>** work on C-style arrays as well.

Different Types of Iterators

Consider:

```
vector<int> v { 1, 2, 3, 4 };
for( auto itr = v.crbegin(); v != v.crend(); ++itr) {
   cout << *itr << " ";
}
```

Outputs:

```
4 3 2 1
```

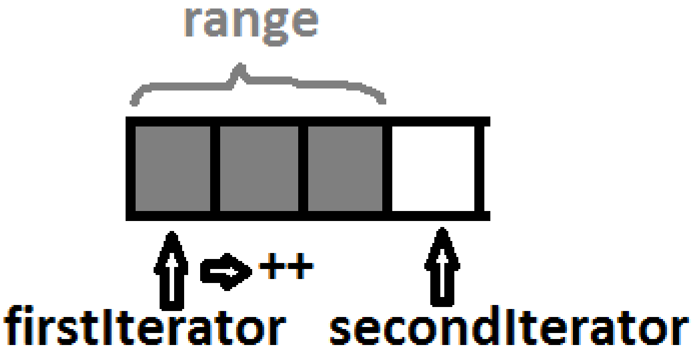
Remarks:

- We use the prefix **++** because it is more efficient (it avoids an extra copy).
- We use **!= v.crend()**, etc., instead of **< v.crend()**, etc. because **<** can only be used for certain types of iterators, but **!=** applies to all of them.

Iterator Ranges

Typically when we work with generic algorithms, if we provide two iterators as input, we mean that we wish to work with values beginning with the first iterator, up to, but not including the second iterator.

someAlgorithm(firstIterator, secondIterator, otherStuff);



std::sort

Recall that with **operator<** defined for elements of a **std::vector**, we can sort the vector by using **std::sort** (found in **<algorithm>**).

```
vector<string> vec { "orange", "apple", "cantaloupe", "apricot"};

// from sort, vec == { "apple", "apricot", "cantaloupe", "orange"}
sort( begin(vec), end(vec) );
```

std::sort

We can also pass **std::sort** a **predicate** (a function that accepts arguments of the data type stored in the container and returns a **bool**). This can be used in lieu of **operator<.**

Suppose we define the function

```
/**
This function determines if the left string is shorter than the right string
@param left the first string being compared
@param right the second string being compared
@return whether the left string is shorter than the right string
*/
bool lessInSize( const string& left, const string& right) {
   return left.size() < right.size();
}
```

Then we can sort the vector of fruits by increasing size, instead!

std::sort

```
// makes vec == { "apple", "orange", "apricot", "cantaloupe"}
sort( vec.begin(), vec.end(), lessInSize );
```

We pass in the function name as an argument and instead of calling **operator<** to determine if **a < b**, **lessInSize(a,b)** is used instead to determine if **"a < b"** (quotes added because it isn't the normal less than operator).

std::sort

Remark: the **std::sort** function only works if the container it is given has **random access iterators**.

It can work for **std::vector**, **std::string**, **std::deque**, or even C-style arrays, but it does not work for **std::list**.

std::find

Provided there is **operator==**, we can use the **std::find** function (found in **<algorithm>**) to

- return an iterator to the first element of the container (in the frame of reference of its first iterator) referencing the value sought or
- return the second iterator given, if no such element is found.

std::find

If the const version of the **begin/end/rbegin/rend** functions are used, the iterators (or pointers) returned reference const variables, protecting them from modification.

```
char array [] = { 'a', 'b', 'c', 'b'};

// will not find element, returns end(array) == array+4
auto iter1 = find( begin(array), end(array), 'd');

// will return rbegin(array), iterator for array+3
auto iter2 = find( crbegin(array), crend(array), 'b');

*iter2 = 'x'; // ERROR: iter2 points to const char!
```

std::find_if

The **std::find_if** function, found in **<algorithm>**, searches through a container to find the first element where a given condition is upheld.

It accepts 3 arguments: one iterator, denoting the start of the search range, a second iterator just past the search range; and a predicate.

The function returns the first iterator within range where the predicate is true, starting from the first iterator, or it returns the second iterator if the predicate is never true over the range.

std::find_if

We could, for example, try to find if there is an int divisible by 7 in a **std::vector<int>**, **v.**

By defining

```
bool divisibleBy7(int i) {
   return ( i % 7 == 0); // if mod is 0, multiple of 7
}
```

We can write:

```
auto iter = find_if( begin(v), end(v), divisibleBy7 );
```

std::for_each

With **std::for_each**, we can pass a range of values covered by two iterators and a function to invoke upon each element of the range. For example, given

```
void negate(int & i) {
   i *= (-1);
}
```

We can negate all elements of a **std::vector<int> v**. If for example, **v == { 2, 4, 6, 8 }** then after

```
for_each( v.begin(), v.end(), negate);
```

we have **v == {-2, -4, -6, -8 }**.

Summary

- The C++ Standard has many built-in containers besides just **std::vectors**.
- A **std::set** stores data uniquely, in a sorted manner, and supports quick insertions/removals.
- A **std::list** can quickly append/remove elements from the front/back and insert/remove an element at a given position.
- A **std::pair** pairs up two values in a simple **struct**.
- Iterators come in many forms through the **begin**, **end**, **cbegin**, **cend**, **rbegin**, **rend**, **crbegin**, and **crend** functions.
- Using the non-member version of **begin**, **end**, etc., is more versatile.
- Many generic algorithms allow us to specify a range of data spanned by iterators to perform a given task, like sorting, and we can sometimes pass function arguments/predicates to give more specialized behaviour.