ECE 2574: Data Structures and Algorithms - STL Containers and Algorithms

C. L. Wyatt

The goal of today's meeting is to review the standard library

- Containers and Iterators
- Algorithms

"The best code is that already written and tested"

The C++ standard library is well-constructed and tested

- prefer to use containers and algorithms from the standard library rather than hand-coded data structures and algorithms.

In this course we saw how to implement data structures and common algorithms for sorting and searching. However, the C++ standard library provides implementations of these that are efficient and well tested, so you should prefer to use them over hand-coded approaches whenever feasible.

std::array is a wrapper around raw arrays

- supports standard access members (at, [], front, back)
- has a size() member
- supports fill and swap
- can be empty
- very low overhead

Example:

```
std::array<int,10> a;
a.fill(1);
assert(a[3] == 1);
assert(a.size() == 10);
```

std::vector is a dynamically sized array-based container

- the most useful linear data structure
- see members size, capacity, and reserve
- grows exponentially
- supports insert - much more efficient than you might think
- watch out for iterator invalidation

Example

```
std::vector<int> v;
std::cout << v.capacity() << std::endl;
for(int i = 0; i < 100; ++i){
    v.push_back(i);
    std::cout << v.capacity() << std::endl;
}
```

std::deque is a dynamically sized double ended queue

- not contiguous in memory
- access either end: push_front or push_back
- generally better performance than std::list
- insert is faster than std::vector

Example:

```
std::deque<int> d;
 for(int i = 0; i < 100; ++i){
   d.push_back(i);
   d.push_front(i);
 }
```

std::list and std::forward_list

- doubly and singly linked-lists respectively
- constant time insertion anywhere
- no random access
- std::list supports bidirectional iteration
- space efficient, no extra space as in std::vector
- can be less efficient than std::vector because of cache misses

adaptors provide wrappers around other containers

- stack (deque)
- queue (deque)
- priority_queue (a heap using vector for storage)

std::map and std::multimap are dictionaries (key,value)

- std::map requires unique keys and value
- implemented as red-black tree (balanced binary tree)
- index operator[] is very handy

Example:

```
std::map<std::string, int> occurances;
occurances["hello"] += 1;
occurances["hello"] += 1;
occurances["goodbye"] += 1;

for(std::map<std::string, int>::iterator it = occurances
    it != occurances.end();
    ++it)
  {
    std::cout << "You said " << it->first << " "
              << it->second << " times." << std::endl;
  }
```

See also std::set and std::multiset (no value, just a key)

Hash tables

- unordered_set / unordered_map
- unordered_multiset / unordered_multimap
- constant (amortized) time find, insert, remove

Same Example

```
std::unordered_map<std::string, int> occurances;
occurances["hello"] += 1;
occurances["hello"] += 1;
occurances["goodbye"] += 1;

for(std::unordered_map<std::string, int>::iterator it = o
    it != occurances.end();
    ++it)
  {
    std::cout << "You said " << it->first << " "
              << it->second << " times." << std::endl;
  }
```

algorithms library

- Non-modifying sequence operations
- Modifying sequence operations
- Partitioning operations
- Binary search
- Set operations
- Heap operations
- min/max
- numeric (see random number generators too)

Example

Consider the following task: find the largest numerical value in a fixed length list of integers.

```
int a[] = {5,9,7,4,41,3,16,11,5};
int len = sizeof(a)/sizeof(a[0]);

assert( max(a,len) == 41 );
```

C-style implementation of max

```
int max(const int a[], const int len)
{
  int m = a[0];
  for(int i = 0; i < len; ++i)
    {
      if(a[i] > m) m = a[i];
    }
  return m;
}
```

C++03 implementation of max

```
int max(const int a[], const int len)
{
  std::vector<int> v(a, a+len);

  int m = v[0];
  for(int i = 0; i < len; ++i)
    {
      if(v[i] > m) m = v[i];
    }
  return m;
}
```

C++03 implementation of max using iterators

```
int max(const int a[], const int len)
{
  std::vector<int> v(a, a+len);

  int m = v[0];
  for(std::vector<int>::iterator it = v.begin();
      it != v.end(); ++it)
    {
      if(*it > m) m = *it;
    }
  return m;
}
```

C11-style implementation of max

```
int max(const int a[], const int len)
{
  std::vector<int> v(a, a+len);

  std::vector<int>::iterator result;
  result = std::max_element(v.begin(), v.end());
  return *result;
}
```

This can be shortened further

```
int max(const int a[], const int len)
{
  return *(std::max_element(a, a+len));
}
```

Example: using for_each rather than an explicit loop

- loops can be a source of bugs
- instead write a functor and use for_each

See example.

Next Actions and Reminders

- Program 5 due Monday

Please, don't forget to take the SPOT survey!