Comp151

STL: Introduction to STL Algorithms

STL Algorithms

- The Standard Template Library not only contains container classes, but also algorithms that operate on sequence containers. To use them, we must write `#include <algorithm>` in our program.
- In this lesson we will see a few different algorithms contained in the STL (for others see the textbook):
  - `sort()` (with and without explicit comparator functions)
  - `find()`, `find_if()`
  - `for_each()`
  - `transform()`, `copy()`
  - `count_if()`

Example: STL Algorithm – `sort()`

- Let `vector<T> A`; for some class `T`.
- Let `vector<T>::iterator p, q`
- `sort(p, q)` sorts `A` between `p` and `q`.
- Common case is `sort(A.begin(), A.end())` sorts all of `A`.

```
// sort without comparators
#include<iostream>
#include<vector>
#include<string>
#include<algorithm>
using namespace std;

template<class Iterator>
void Display(Iterator start, Iterator end)
{
    for( Iterator p = start; p != end; ++p )
    cout << *p << " ";
}

int main()
{
    vector<string> composer;
    composer.push_back("Mozart"); composer.push_back("Bach");
    composer.push_back("Chopin"); composer.push_back("Beethoven");
    cout << "composer: "; Display(composer.begin(), composer.end()); cout << endl;
    sort(composer.begin(), composer.end());
    cout << "composer: "; Display(composer.begin(), composer.end()); cout << endl;

    vector<int> v;
    for (int i = 1; i < 13; i++) {
        v.push back(i*i % 13);
    }
    cout << "v: "; Display(v.begin(), v.end()); cout << endl;
    sort(v.begin(), v.end());
    cout << "v: "; Display(v.begin(), v.end()); cout << endl;
}
```

Output

```
composer: Mozart Bach Chopin Beethoven
composer: Bach Beethoven Chopin Mozart
v: 1 4 9 3 12 10 10 12 3 9 4 1
v: 1 1 3 3 4 4 9 9 10 10 12 12
```

Example: STL Algorithm – `sort()`

- Let `vector<T> A`; for some class T.
- Let `vector<T>::iterator p, q`
- `sort(p,q)` sorts A between p and q.
- Common case is `sort(A.begin(), A.end())` sorts all of A.
- `sort()` also works with `deque` objects but not with list objects.
- In general, `sort()` works with any random access sequence container.
- Guaranteed O(*n* log *n*) running time.

Another Example: STL Algorithm – `find()`

```
#include <algorithm>
#include <string>
#include <list>

int main()
{
    list<string> composer;
    composer.push_back("Mozart"); composer.push_back("Bach");
    composer.push_back("Chopin"); composer.push_back("Beethoven");

    list<string>::iterator p;
    p = find(composer.begin(), composer.end(), "Bach");

    if (p == composer.end()) {
        cout << "Not found." << endl;
    } else if (++p != composer.end()) {
        cout << "Found before: " << *p << endl;
    } else {
        cout << "Found at the end." << endl;
    }
}
```

Algorithms, Iterators, and Sub-sequences

- Sequences/Sub-sequences are specified using <u>iterators</u> that indicate the beginning and the end for an algorithm to work on.
- Here we find the <u>2nd</u> occurrence of the value, 341, in a sequence.

```
// File "init.cpp"
int f(int x) { return -x*x + 40*x + 22; }
// 22 61 98 133 166 197 226 253 278 301 322 341 358 373 386 397
// 406 413 418 421 422 421 418 413 406 397 386 373 358 341 322 301

template<typename T>
void my_initialization(T& x)
{
    const int N = 39;
    for (int j = 0; j < N; ++j) {
        x.push_back( f(j) );
    }
}
```

Example: Algorithm with Iterators & Sub-Sequence

```
#include <vector>
#include <algorithm>
#include "init.cpp"

int main()
{
    const int search_value = 341;
    vector<int> x; my_initialization(x);

    vector<int>::iterator p;
    p = find(x.begin(), x.end(), search_value);

    if (p != x.end()) {                                  // Value found!
        p = find(++p, x.end(), search_value);            // Find again
        if (p != x.end()) {                              // Value found again!
            cout << "Found after: " << *--p << endl;
        }
    }
}
```

STL `find()` – ‘Implementation’

```
template<class IteratorT, class T>
IteratorT find(IteratorT first, IteratorT last, const T& value)
{
    while (first != last && *first != value) {
        ++first;
    }
    return first;
}
```

- `find()` searches linearly through a sequence, and stops when an item matches the 3rd argument.
- A big limitation of `find()` is that it requires an <u>exact</u> match by <u>value</u>.