**Comp151**

**STL: Algorithms**



## STL Algorithms

The Standard Template Library not only contains container classes, but also algorithms that operate on sequence containers. To use 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 appendix in *Lippman and Lajoe*):

- `sort()` (with and without explicit comparator functions)
- `find()`, `find_if()`
- `for_each()`
- `transform()`, `copy()`
- `count_if()`

We will also learn the basics of *function pointers* and *function objects*.



## 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

```
//sort without comparators
#include<iostream>
#include<vector>
#include<string>
#include<algorithm>

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> L;
    for(int i = 1; i < 13; i++)
        L.push_back(i*i % 13);

    cout << "L: "; Display(L.begin(), L.end()); cout << endl;
    sort(L.begin(), L.end());
    cout << "L: "; Display(L.begin(), L.end()); cout << endl;
}
```

```
OUTPUT
------
L: 1 4 9 3 12 10 10 12 3 9 4 1
L: 1 1 3 3 4 4 9 9 10 10 12 12
```

## 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 iterators that indicate the beginning and the end for an algorithm to work on.

Here we find the *2nd* 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(vector<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 Iterator, class T>
Iterator find(Iterator first, Iterator 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 *exact* match by value.

## Generic Algorithms with Function Arguments



Let's search in a container for a value that satisfies a boolean condition *specified by a C++ function.*

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

bool greater_than_350(int value) { return value > 350; }

int main()
{
    vector<int> x; my_initialization(x);
    vector<int>::iterator p = find_if( x.begin(), x.end(), greater_than_350 );

    if (p != x.end())
        cout << "Found element:   " << *p << endl;
}
```

## STL Algorithms — find_if()



```
template <class Iterator, class Predicate>
Iterator find_if(Iterator first, Iterator last, Predicate pred)
{
    while (first != last && !pred(*first))
        ++first;
    return first;
}
```

- `find_if()` is a more general algorithm than `find()` in that it stops when a condition is satisfied.
- This allows partial match, or match by keys.
- The condition is specified by a function.

## C Function Pointer



- Inherited from C, C++ allows a function to be passed as argument to another function.
- Actually, we say that we pass the function pointer.
- e.g. if you do "`man 3 qsort`" on Unix:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compare)(const void *, const void *))
```

i.e. The 4th argument is a function pointer, whose type is:

```
int (*)(const void*, const void*)
```

e.g. `int compare(int* i, int* j) { return (*i) - (*j); }`

## Example: C Function Pointer



```
#include <iostream>

int max(int x, int y) { return (x > y) ? x : y; }
int min(int x, int y) { return (x > y) ? y : x; }

void main()
{
    int (*f)(int x, int y);
    int choice;

    cin >> choice;
    if (choice == 1)
        f = max;
    else
        f = min;

    cout << f(3,5) << endl;
}
```

```
#include<iostream>
#include<vector>
#include<string>
#include<algorithm>

class Person
{ public:
  string name;
  int    ID;
  Person(string n, int id): name(n), ID(id){};
};

void Display(vector<Person>  & V)
{ vector<Person>::iterator p;
  for(  p = V.begin(); p != V.end(); p++)
     cout << "(" << (*p).name  << "," << (*p).ID << ")  ";
  cout << endl;}
```

## STL sort() – Again



The STL sort() function seen before can actually accept a function as its third argument:

```
template<class Iterator, class predicate>
void sort(Iterator first, Iterator last, Class  pred>
```

It sorts everything between *first* and *last* using the predicate *pred* as a comparison function.

```
bool lt_name(const Person &P1, const Person & P2)
{return (P1.name < P2.name);}

bool lt_ID(const Person &P1, const Person & P2)
{return (P1.ID < P2.ID);}

int main()
{ vector<Person> Fac;
  Fac.push_back(Person("Kafeza", 20));
  Fac.push_back(Person("Golin", 60));
  Fac.push_back(Person("Wood", 50));
  Fac.push_back(Person("Shen", 40));
  Fac.push_back(Person("Tang", 35));

  Display(Fac);
  sort(Fac.begin(), Fac.end(), lt_name); Display(Fac);
  sort(Fac.begin(), Fac.end(), lt_ID);  Display(Fac);}
```

```
OUTPUT
------
(Kafeza,20)  (Golin,60)  (Wood,50)  (Shen,40)  (Tang,35)
(Golin,60)  (Kafeza,20)  (Shen,40)  (Tang,35)  (Wood,50)
(Kafeza,20)  (Tang,35)  (Shen,40)  (Wood,50)  (Golin,60)
```

## Example: STL Algorithm — for_each()



```
#include <vector>
#include <algorithm>
#include "init.cpp"
void Print( int val) { cout << val << endl;}

int main()
{   vector<int> x; my_initialization(x);
    for_each(x.begin(), x.end(), Print);
}

OUTPUT
-------
22
61
.
```

## STL Algorithms — for_each()



```
template <class Iterator, class Function>
Function for_each(Iterator first, Iterator last, Function g)
{
    for ( ; first != last; ++first)
        g(*first);
    return g;
}
```

*for_each* calls function *g()* on every element in the container between *first* and *last*.

## Function Objects

STL has a more generalized concept of function pointer: Any "object" that can be "called" is a function object; function pointers are just one example.

- An object can be called if it supports the `operator()`.

```
// File "greater_than.h"
class Greater_Than
{ private:
    int limit;
  public:
    Greater_Than(int a) : limit(a) { }
    bool operator()(int value) { return value > limit; }
};
```



## More Function Objects

```
// File "greater_than.h"
class Greater_Than
{ private:
    int limit;
  public:
    Greater_Than(int a) : limit(a) { }
    bool operator()(int value) { return value > limit; }
};
```

`Greater_Than` is a *class*.
`Greater_Than gt_five(5);` creates an object named *gt_five* of class *Greater_Than* that is constructed with parameter a=5. This means that, inside *gt_five*, `limit=5`.

Now notice that `operator()` has been overloaded so `gt_five(value)` is defined to be afunction that returns `(value >5)`.

```
//Greater_Than.cpp
#include<iostream>
class Greater_Than
{ private:
    int limit;
  public:
    Greater_Than(int a) : limit(a) { }
    bool operator()(int value) { return value > limit; }
};

main()
{ Greater_Than gt_five(5), gt_ten(10);

 if (gt_five(7))
     cout <<"7 > 5" << endl;
 else cout << "7 <=5" << endl;

 if (gt_ten(7))
     cout <<"7 > 10" << endl;
 else cout << "7 <=10" << endl;
}
```

## How to Use Function Objects in STL Algorithms?



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

int main()
{ Greater_Than g(350);
    vector<int> x; my_initialization(x);
    vector<int>::iterator p;

    p = find_if( x.begin(), x.end(), g );

    if (p != x.end())
        cout << "Found element " << *p << endl;
}
```

## How to Use Function Objects STL Algorithms? ...



When `find_if()` examines each item, say `x[j]` in the container `vector<int> x`, the `Greater_Than` function object `g` will be called using its `operator()` with the container item. i.e.

```
g(x[j]) // Or  formally: g.operator()(x[j])
```

Since `g(i)==1` if and only if $i \geq 350$ this will find the first item in `vector<int> x` that is $\geq 350$.

## A sneakier way of writing the same thing



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

int main()
{
    vector<int> x; my_initialization(x);
    vector<int>::iterator p =
        find_if( x.begin(), x.end(), Greater_Than(350) ); // <—

    if (p != x.end())
        cout << "Found element " << *p << endl;
}
```

## What does this mean?



```
find_if( x.begin(), x.end(), Greater_Than(350) );
```

In this case the `Greater_Than(350)` is actually a *constructor*. It is equivalent to writing `Greater_Than g(350);` and then writing

```
find_if( x.begin(), x.end(), g );
```

When `find_if` is running it will call `g(x[j])` for the items in `vector<int> x`.

## Function objects ...



An object that can be called like a function is called a function object, functoid, or functor.

- Function objects are more powerful than functions, since they can have data members and therefore carry around information or *internal states*.
- A function object must have at least the operator() overloaded so that it can be called.
- A function object (or a function) that returns a boolean value (of type `bool`) is called a predicate.

## Example: STL Algorithm — count_if()

Here we count the number of elements that are larger than 10:

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

int main()
{
    vector<int> x;
    my_initialization(x);
    int num = count_if( x.begin(), x.end(), Greater_Than(10) );
}
```

In line

```
for_each( x.begin(), x.end(), Sum() );
```

the `Sum()` is a constructor that creates a local object of class Sum. Call this local copy `s`. The `for_each` runs $s(x[j])$ for each $j$. So, at the end, `s.sum` contains the value of the sum of all of the items in `x`.

Referring back to the template of `for_each` we see that it returns a copy of `s`. The line

```
Sum S = for_each( x.begin(), x.end(), Sum() );
```

calls a copy constructor which makes `S` into a copy of that copy of `s`, so `S` is a copy of `s` and `S.sum` contains the value of the sum of all of the items in `x`.



## Example: STL Algorithm — for_each() to Sum

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

class Sum {
  private:
    int sum;
  public:
    Sum() : sum(0) { }
    void operator()(int value) { sum += value; }
    int result() const { return sum; }
};

int main()
{
    list<int> x; my_initialization(x);
    Sum S = for_each( x.begin(), x.end(), Sum() );
    cout << "Sum = " << S.result() << endl;
}
```

To confuse matters, the code

```
int main()
{
    list<int> x; my_initialization(x);
    Sum S;
    for_each( x.begin(), x.end(), S);
    cout << "Sum = " << S.result() << endl;
}
```

would return 0! The reason that it doesn't return the expected value is that `for_each` calls its arguments *by value*. This means that the total sum is stored in a *local copy* of `S` and not in `S` itself. Therefore `S` iself never changes from its initially constructed value so `S.sum`=0.

## STL Algorithms — transform()



```
template <class Iterator1, class Iterator2, class Function>
Iterator2 transform(Iterator1 first, Iterator1 last, Iterator2 result, Function g)
{
    for ( ; first != last; ++first, ++result)
        *result = g(*first);
    return result;
}
```

*transform* will apply function *g()* to all of the items in the sequence between *first* and *last*. The resulting sequence is written to the location 'pointed' to by *result*.

## Example: STL Algorithm — for_each() to Add



```
// File: "Add.h"
#include <list>
#include <vector>
#include <algorithm>

class Add {
  private: int data;
  public:
    Add(int i) : data(i) { }
    int operator()(int value) { return value + data; }
};

void Print (int val) {cout << val << " ";}
```

## Example: STL Algorithm — for_each() to Add ...



```
#include "Add.h"

int main()
{
    list<int> x;
    for(int i=1; i <10; i++)
      x.push_back(i);
    vector<int> y(x.size());

    transform( x.begin(), x.end(), y.begin(), Add(10) );
    for_each( y.begin(), y.end(), Print);
    cout << endl;
}
```

## Other Algorithms in the STL



- `min_element` and `max_element`
- `equal`
- `generate` (Replace elements by applying a function object)
- `remove`, `remove_if` Remove elements
- `reverse`, `rotate` Rearrange sequence
- `random_shuffle`
- `binary_search`
- `sort` (using a function object to compare two elements)
- `merge`, `unique`
- `set_union`, `set_intersection`, `set_difference`

Documentation for SGI's STL implementation can be found at:
`http://www.sgi.com/Technology/STL/doc_introduction.html`.