Comp151

STL: Function Objects or Functors

Function Objects, or Functors

- STL has a more generalized concept of function pointer: Any "object" that can be "called" is a function object or functor.
- C function pointers are just one example.
- C++ gives more options: any object can be "called" if it supports `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; }
};
```

- This runs at least as fast as using function pointers – and sometimes faster! (Why?)

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 a function 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;
    }
}
```

Function Objects Can Carry State

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

- Function objects can carry state.
- E.g., `limit`=5 is the state of the `gt_five` object.
- Big advantage – since you cannot do this using simple C-style function pointers!

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 in 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 >= 350` this will find the first item in `vector<int> x` that is `>= 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. This 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 or functor (also sometimes functoid, but this often carries other meanings not in STL).
- 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.