Comp151

STL: Function Pointers

Function Pointers

- Recall that a big limitation of `find()` is that it requires an exact match by value.
- Function pointers are the key to removing this limitation. This dramatically increases the power of STL's generic algorithms.
- We will learn the basics of function pointers.
- Later, we will learn about an even more powerful generalization called function objects.

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"
using namespace std;

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 IteratorT, class PredicateT>
IteratorT find_if(IteratorT first, IteratorT last, PredicateT 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

- C++ allows a function to be passed as argument to another function. (This is a tradition that is inherited from C.)
- What's actually happening is that we pass a function pointer: a pointer to the memory address of the executable code.
- e.g. if you type `man 3 qsort` on Unix, you'll see:

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

  - This means the 4th argument, `compare`, is a function pointer, whose type is:

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

  - For example, you could pass a pointer to the following function:

```
int compare_floats(float* i, float* j)
{
    return (*i) – (*j);
}
```

Example: C Function Pointer

```
#include <iostream>
using namespace std;

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;
}
```

STL sort() – Again

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

```
template<class IteratorT, class PredicateT>
void sort(IteratorT first, IteratorT last, PredicateT pred>
```

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

```
#include<iostream>
#include<vector>
#include<string>
#include<algorithm>
using namespace std;

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

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

```
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> people;
    people.push_back(Person(‘‘K", 20));
    people.push_back(Person("G", 60));
    people.push_back(Person("W", 50));
    people.push_back(Person("S", 40));
    people.push_back(Person("T", 35));

    Display(people);
    sort(people.begin(), people.end(), lt_name); display(people);
    sort(people.begin(), people.end(), lt_id); display(people);
}
```

Output

```
(K,20) (G,60) (W,50) (S,40) (T,35)
(G,60) (K,20) (S,40) (T,35) (W,50)
(K,20) (T,35) (S,40) (W,50) (G,60)
```

Example: STL Algorithm – `for_each( )`

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "init.cpp"
using namespace std;

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 IteratorT, class FunctionT>
FunctionT for_each(IteratorT first, IteratorT last, FunctionT g)
{
    for ( ; first != last; ++first) {
        g(*first);
    }
    return g;
}
```

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