Reminders:

• Online course evaluations

• Reading: Chapter 10 (finish)

Last class:

• Class Templates

• STL intro
  • Containers

Today:

• STL (finish)
  • Iterators
  • Algorithms (search, etc.)
  • Examples

• Assorted topics:
  • Function objects (functors)
  • Multiple Inheritance
  • Polymorphism
  • gdb and valgrind
**STL Iterators**

What are iterators and what are they good for?

- Objects pointing to other objects (in containers)
- Interface between a container and an algorithm

Getting an iterator from a container:

- `iterator begin();`
- `const_iterator begin() const;`
- `iterator end();`
- `const_iterator end() const;`

Remember, iterator types are specific to the object they “point” to (just like Java).

(Some) Iterator operations:

- `++iter`
- `*iter`
- `operator==`

An example:

```cpp
list<double> L;
L.push_back(1.2);
L.push_front(3.4);
L.insert(L.begin(), 5.6);
L.insert(L.end(), 7.8);

list<double>::const_iterator iter;
for (iter = L.begin(); iter != L.end(); ++iter)
    cout << *iter << " ";
cout << endl;
```
Iterator Kinds and Operations

All iterators support ++ (both prefix and postfix), *, ==

    forward_list

Bidirectional Iterators support --

    list, set, map

Random Access Iterators support +=k

    vector, deque, array

Input/Output Iterators
Function Objects (Functors)

Classes that define `operator()`

Create objects that are basically functions

Like function pointers but can have state

Main use: As arguments to STL algorithm functions

Example (simple use):

```cpp
class MultiplyBy {
    public:
        MultiplyBy(double f): factor(f) {}

        double operator() (double val) const {
            return val * factor;
        }

    private:
        double factor;
}

// in main():

MultiplyBy doubler(2.0);
double x = 5.0;
double y = doubler(x);
cout << x << " doubled is " << y << endl; // prints: 5 doubled is 10
```
STL Algorithms

STL provides algorithms that work on containers, yet are independent of container implementations or details.

#include <algorithm>

Interfacing is done through iterators (#include <iterator>)

Common algorithms:

- find
- count, count_if
- sort, stable_sort
- transform
- all_of, any_of, none_of
- partition
- copy
- for_each
- generate

```cpp
int numbers[] = {3, 5, 1, 2, 4};
vector<int> V(numbers, numbers+5);
vector<double> W(W.size());

sort(V.begin(), V.end());
copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
cout << endl;  // prints: 1 2 3 4 5

transform(V.begin(), V.end(), W.begin(), MultiplyBy(1.5));
copy(W.begin(), W.end(), ostream_iterator<int>(cout, " "));
cout << endl;  // prints: 1.5 3 4.5 6 7.5

vector<int>::iterator found;
found = find_if(V.begin(), V.end(), isEven());
cout << *found << endl;  // prints: 2
```
Multiple Inheritance

Deriving/Inheriting from more than one class

Complicates design, introduces ambiguities, makes debugging much harder; use with extreme care and understand the rules

The Diamond problem

Polymorphism

A pointer to a derived class is type-compatible with a pointer to its base class

Accessing members

virtual functions (in particular, the destructor)
Debugging Tools

gdb:

valgrind:
CS368: The End

Evaluations

Have a good winter!