Ve 280

Programming and Introductory Data Structures

Function Objects;

Standard Template Library: Sequential Containers

Outline

- Function Objects

- STL Sequential Container: `vector`
  - Some Basic Operations
  - Iterator
  - Operations with Iterator

Review: Motivation of Function Objects

- We want to write a generic "has_a" function using function pointers
- **<u>Question</u>**: How would you use `has_a` to see if a list `l` has any elements larger than 2? How would you use it to see if a list has any elements larger than 42?

```
bool has_a(list_t l, bool (*pred)(int)) {
  if(list_isEmpty(l)) return false;
  else if(pred(list_first(l))) return true;
  else return has_a(list_rest(l), pred);
}
```

Review: Motivation of Function Objects

- One way to solve the problem of creating a larger_than function with function pointers requires a global variable:

```
int larger_target;  // global variable
bool larger(int n) {
  return (n>larger_target);
}
```

- To use this, we would do something like this:

```
list_t l;
...
cin >> larger_target; // get upper bound
cout << has_a(l, larger);
```

Motivation

- To avoid using a global variable, we want a "function-creating function" – one that, given an integer `i`, returns a predicate that takes one integer argument, N, and returns true of N > `i`.

- There is no way to do this with C++ functions

- But, we **can** do it with the class mechanism plus one neat trick – **operator overloading**

Overloading Function Call

- Recall that we can overload many operators of a class. E.g.,

  `Foo &operator=(const Foo &f);`

- It turns out that the "function-call" is just another operator and we can overload it.
- Suppose we have a class called `Multiply2`, with no private members and only one public one:

```
class Multiply2 {
 public:
   int operator() (int n);
};
```

Overloading Function Call

```
class Multiply2 {
 public:
   int operator() (int n);
};
```

- This public method overloads the "function-call" operator on **instances** of `Multiply2` – the method takes a single integer argument, and returns an integer result
- Here's the body of that method:

```
int Multiply2::operator() (int n) {
   return n*2;
}
```

The function-call method takes an integer argument and returns twice that argument.

Function Objects

- So, we can do this:

```
Multiply2 doubleIt;
cout << doubleIt(4) << endl;
```

- The second line looks like a function call; however, `doubleIt` is **not** a function. Rather, it is an instance of the class `Multiply2`.
- The class `Multiply2` has defined the "function-call operator". So, we invoke that operator, passing the argument 4.
- The method body returns 2*4, printing 8 to the terminal.
- Objects that overload the function-call operator are called **function objects**, or sometimes **functors**.

Function Objects

Implementation

- So far, this is not very interesting
- However, unlike functions, objects can have **per-object state**, which allows us to specialize on a per-object basis
- For example, suppose we defined the class `MultiplyN` to be:

```
class MultiplyN {
  int factor;
  MultiplyN() {}  // Private ctor
 public:
  MultiplyN(int n);
  int operator()(int n);
};
```

The idea here is that when we create instances of `MultiplyN`, we bind the "multiplicative factor" to some constant, and later can multiply numbers by that factor.

Function Objects

Implementation

- So far, this is not very interesting
- However, unlike functions, objects can have **per-object state**, which allows us to specialize on a per-object basis
- For example, suppose we defined the class `MultiplyN` to be:

```
class MultiplyN {
   int factor;
   MultiplyN() {}  // Private ctor
 public:
   MultiplyN(int n);
   int operator()(int n);
};
```

Private ctor ensures that you must construct the object using the version **`MultiplyN(int n)`**, because we need to assign initial value for **`factor`**

Function Objects

Implementation

- So, the constructor would be:

```
MultiplyN::MultiplyN(int n) {
   factor = n;
}
```

- And the function-call operator:

```
int MultiplyN::operator() (int n) {
   return n*factor;
}
```

Function Objects

Implementation

- Now, we can use this new class to "generate" specialized functors:

Note the different use of "()"

```
MultiplyN doubleIt(2);
MultiplyN tripleIt(3);
cout << doubleIt(4) << endl;
cout << tripleIt(4) << endl;
```

This () calls the constructor

This () calls function-call operator

- Which prints to the screen:

```
8
12
```

Function Objects

Implementation

- Finally, we can use functors to write our generic `has_a` routine.
- We first define an abstract `Predicate` class, specifying that a `Predicate` must provide an appropriate function-call method:

```
class Predicate {
 public:
  virtual bool operator()(int n) = 0;
};
```

Function Objects

Implementation

- Now, define `has_a` in terms of this `Predicate` class:

```
bool has_a(list_t l, Predicate &pred) {
  if(list_isEmpty(l)) return false;
  else if(pred(list_first(l)))
    return true;
  else return has_a(list_rest(l), pred);
}
```

<u>**Note**</u>: The body of `has_a` is **exactly the same** as it was before. But, rather than take a function pointer, it takes a functor.

Function Objects

Implementation

- If we want to use `has_a` to check for entries greater than some specific value, we can write a subtype of `Predicate`:

```
class GreaterN : public Predicate {
   int target;
   GreaterN() {}
 public:
   GreaterN(int n);
   bool operator() (int n);
};
GreaterN::GreaterN(int n) {
   target = n;
}
bool GreaterN::operator() (int n) {
   return (n > target);
}
```

Function Objects

Implementation

- We can also check for entries less than some specific value by writing another subtype of `Predicate`:

```
class LessN : public Predicate {
    int target;
    LessN() {}
  public:
    LessN(int n);
    bool operator() (int n);
};
LessN::LessN(int n) {
    target = n;
}
bool LessN::operator() (int n) {
    return (n < target);
}
```

Function Objects

Implementation

- Now, given a list, we can find out if it has elements greater than 2:

```
list_t l;
... // fill in the list
GreaterN gt2(2);
cout << has_a(l, gt2);
```

- or 42:

```
GreaterN gt42(42);
cout << has_a(l, gt42);
```

Function Objects

Implementation

- We can also test if a list has values less than 2:

```
list_t l2;
... // fill in the list
LessN lt2(2);
cout << has_a(l2, lt2);
```

- or 42:

```
LessN lt42(42);
cout << has_a(l2, lt42);
```

Function Objects

Implementation

- We can even get limits from the user:

```
list_t l;
... // fill in the list

int GT_Limit, LT_Limit;
cin >> GT_Limit >> LT_Limit; // user input

GreaterN gt(GT_Limit);
LessN lt(LT_Limit);

cout << has_a(l, gt);
cout << has_a(l, lt);
```

So, the ability of objects to carry per-object state **plus** override the "function call" operator gives us the equivalent of a "function factory".

Function Objects

Implementation

- We can even get limits from the user:

```
list_t l;
... // fill in the list

int GT_Limit, LT_Limit;
cin >> GT_Limit >> LT_Limit; // user input

GreaterN gt(GT_Limit);
LessN lt(LT_Limit);

cout << has_a(l, gt);
cout << has_a(l, lt);
```

This allows us to generalize predicates without resorting to the global variable trick.

Outline

- Function Objects

- STL Sequential Container: `vector`
  - Some Basic Operations
  - Iterator
  - Operations with Iterator

Standard Template Library (STL)
Overview

- We have talked about containers
  - C++ has a **standard template library (STL)** that provides us with an easy way to define containers

- STL defines powerful, template-based, reusable components that implements common data structures and algorithms

- Divided into three components:
  - Containers: data structures that hold a collection of objects of a specified type
  - Iterators: used to examine and nevigate container elements
  - Algorithms: searching, sorting and many others

Containers in STL

- The STL provides three kinds of containers:
  - **Sequential Containers**: let the programmer control the order in which the elements are stored and accessed. The order does not depend on the values of the elements
  - **Associative Containers**: store elements based on their values. The order depends on the value of the elements
  - **Container Adapters**: take an existing container type and make it act like a different type

Sequential Containers

- There are three sequential containers:
  - `vector`: based on arrays.
    - Supports fast random access.
    - Fast insert/delete at the back. Inserting or deleting at other position is slow.
  - `deque` (double-ended queue): based on arrays.
    - Supports fast random access.
    - Fast insert/delete at front or back.
  - `list`: based on a doubly-linked lists
    - Supports only bidirectional **sequential** access.
    - Fast insert/delete at any point in the list.

Vector

- `vector` is a widely used STL container
  - A collection of objects of a **single** type, each of which has an associated integer index.
  - We can create a vector of ints, a vector of strings, etc.

- To use a vector, include the appropriate header and namespace.

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

Vector

- vector is a template. We need to specify the type of objects the vector contains.

```
vector<int> ivec; // holds ints
vector<IntSet> isvec; // holds IntSets
```

Initializing Vector

- `vector<T> v1;`
  - Construct an **empty** vector `v1` that holds objects of type `T`
  - E.g., `vector<int> v1;`
- `vector<T> v2(v1);`
  - Copy constructor.
  - E.g., `vector<int> v2(v1);`
- `vector<T> v3(n, t);`
  - Construct `v3` that has `n` elements with value `t`.
  - E.g., `vector<int> v3(10, -1);`
  - `vector<string> v4(2, “abc”);`

Size of Vector

- `v.size() // number of  elements in v`
- `size()` return a value of `size_type` corresponding to the vector type.
- `vector<int>::size_type`
  - A **companion type** of vector
  - Essentially an unsigned type (unsigned int or unsigned long)
  - **Note**: not `vector::size_type`
- Why companion types?
  - To make the type machine-independent

Size of Vector

- Generally, you can convert `size_type` into `unsigned int`

```
unsigned int s = v.size();
```

- However, using `int` is not recommended

```
int s = v.size(); // not good
```

- If you only want to know whether the vector is empty or not, you can use
  - `v.empty() // true if v is empty`

Add/Remove Element to/from Vector

- Add: `v.push_back(t)`
  - Add element with value `t` to **end** of `v`
- Example

```
vector<int> v;
for(int i = 0; i <5; i++)
     v.push_back(i);
// v is 0,1,2,3,4
```

- Remove: `v.pop_back()`
  - Remove the last element in `v`. No argument. Returns void. `v` must be non-empty

Container Elements Are Copies

- There is no relationship between the element in the container and the value from which it was copied.
- What is the value of `v[0]`?

```
vector<int> v;
int a = 3;
v.push_back(a); // v[0] is 3 now
a = 5; // What is v[0] now?
```

- Subsequent changes to the value that was copied have no effect on the element in the container, and vice versa.

Subscripting Vector

- `v[n]`: returns element at position `n` in `v`

```
vector<int>::size_type ix;
for(ix=0; ix!=ivec.size(); ++ix)
 ivec[ix]=0;
```

- Subscripting does not add elements.

```
vector<int> ivec; // empty vector
for(vector<int>::size_type ix=0; ix!=10; ++ix)
  ivec[ix] = ix; // Error!
```

- An element must exist in order to subscript it.

Good Practice

```
vector<int>::size_type ix;
for(ix=0; ix!=ivec.size(); ++ix)
  ivec[ix]=0;
```

- **<u>Note</u>**: we call the `size` member in the `for` rather than calling it once before the loop and remembering its value.
- Why?
  - Because vector can grow dynamically by adding new elements
  - By putting `size` in `for`, we test on the most current size. It is safer.
- Will it be slow?
  - No! size() is an inline function
  - Inline function: expanded "in line". Avoid function call overhead.

Other Basic Operations on Vector

- `v1 = v2` //replace elements in v1 by a copy of
  // elements in v2
- `v.clear()` // makes vector v empty
- `v.front()` // Returns a reference to the first element
  // in v. v must be non-empty!
- `v.back()` // Returns a reference to the last element in v.
  // v must be non-empty!

Outline

- Function Objects

- STL Sequential Container: `vector`
  - Some Basic Operations
  - Iterator
  - Operations with Iterator

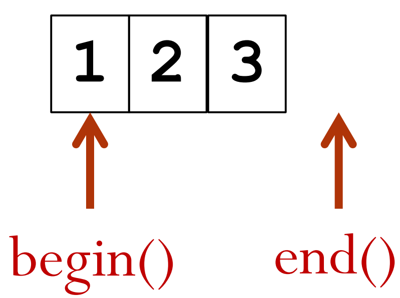
Iterators

- Each container type has a companion **iterator** type.
  - It lets us examine elements and navigate in the container.
- Iterators are more general than subscripts: All of the library containers define iterator types, but only a few of them support subscripting.
- Declare an iterator for vector:
  - E.g., `vector<int>::iterator it;`

- An iterator is a generalization of pointer.
  - They are pointers to the elements of containers.

How to Link Iterator to Vector?

- Use two member functions `begin()` and `end()` of vector
- `v.begin()` returns an iterator pointing to the first element of vector
  - `vector<int>::iterator it = v.begin();`
- `v.end()` returns an iterator positioned **one-past-the-end** of the vector
  - It does not denote an actual element in vector



end()

- `v.end()` is used to indicate when we have processed all the elements in vector

- If the vector is empty, the iterator returned by `begin` is the same as the iterator returned by `end`

Operations on Iterator

- Dereference operator
  - `*iter`: let us access the element to which the iterator refers
  - You can **read**/**write** through `*iter`
- Increment/decrement operator
  - `++iter`, `iter++`: advance to the next item in vector
  - `--iter`, `iter--`: go back to the previous item

Note: you cannot dereference or increment iterator returned by end()

- `iter == iter2` and `iter != iter2`: test whether two iterators point to the same data item

Example

- Sum all the elements of the `vector<int> ivec`.

```
int sum = 0;
vector<int>::iterator it;
for(it=ivec.begin(); it != ivec.end(); ++it)
  sum += *it;
```

- **Question**: what happens when `ivec` is empty? what is the sum?
- Why using iterator instead of subscripting?
  - All container types have associated iterator types, but not all of them have subscripting.

const_iterator

- Using iterator could change the values in the vector.
- `const_iterator` is another iterator type. However, **cannot** use it to change the value.
  - It can only be used for reading, but not writing to, the container elements …
  - … because dereferencing a const_iterator is a const object.
  - Note: its own value can be changed, e.g., we can increment it.

```
vector<string>::const_iterator it;
for(it=text.begin(); it!=text.end(); ++it) {
  cout << *it << endl; // fine
  *it = " "; // error: *it is const
}
```

Iterator Arithmetic

- vector supports iterator arithmetic
  - Not all containers support iterator arithmetic
- `iter+n`, `iter-n`
  - `n` is an integral value
  - adding (subtracting) a value `n` to (from) an iterator yields an iterator that is `n` positions forward (backward)
- We can use iterator arithmetic to move an iterator to an element directly
  - Example: go to the middle

```
vector<int>::iterator mid;
mid = vi.begin() + vi.size()/2;
```

Relational Operation on Iterator

- `>`, `>=`, `<`, `<=`
- E.g., `while(iter1 < iter2)`
- One iterator is less than (<) another if it refers to an element whose position in the container is **ahead** of the one referred to by the other iterator.

- To compare, iterators must refer to elements in the **same** container or one past the end of the container (i.e., `c.end()`).

Outline

- Function Objects

- STL Sequential Container: `vector`
  - Some Basic Operations
  - Iterator
  - Operations with Iterator

Initializing with a Range of Elements

- `vector<T> v(b, e);`
  - Create vector `v` with a copy of the elements from the range denoted by iterators `b` and `e`
- **Note**: **iterator range** is denoted by a pair of iterators `b` and `e` that refer to two elements, or to one past the last element, in the same container.
  - **Note**: the range includes `b` and each element **up to but not including** `e`.
  - It is denoted as `[b, e)`
  - If `b = e`, the range is empty
  - If `b=v.begin()`, `e=v.end()`, the range includes all the elements in `v`

Initializing with a Range of Elements

- We can use this form of initialization to copy only a subsequence of the other container
- Example

```
// assume v is a vector<int>
vector<int>::iterator mid;
mid = v.begin()+ v.size()/2;

// front includes the 1st half of v, from begin
// up to but not including mid
vector<int> front(v.begin(), mid);

// back includes the 2nd half of v from mid
// to end
vector<int> back(mid, v.end());
```

Initializing with a Range of Elements

- `vector<T> v(b, e);`

- We can even use another container type to initialize

```
deque<string> ds(10, "abc");
vector<string> vs(ds.begin(), ds.end());
```

Initializing with a Range of Elements

- Since pointers are iterators, the iterator range can also be a pair of pointers into a built-in array

```
int a[] = {1, 2, 3, 4};
unsigned int sz = sizeof(a)/sizeof(int);
vector<int> vi(a, a+sz);
```

- Note
  - **`sizeof(obj)`**, **`sizeof(type name)`**: return the size in bytes of an object or type name
  - If **`obj`** is an array name, **`sizeof(obj)`** is the total size in byte in that array
- <u>Question</u>: what is the value of **`sz`**?

Initializing with a Range of Elements

```
int a[] = {1, 2, 3, 4};
unsigned int sz = sizeof(a)/sizeof(int);
vector<int> vi(a, a+sz);
```

- `a` points to the first element in array `a`
- `a+sz` points to the location one past the end of array `a`
- Thus, the entire array `a` is copied

Another Way to Add Value: insert()

- `v.insert(p,t)`
  - Inserts element with value `t` **before** the element referred to by iterator `p`.
  - Returns an iterator referring to the element that was added.
- We can use insert to insert at the beginning of vector

```
vector<int> iv(2, 1);
iv.insert(iv.begin(), -1);
```

- We can also insert at the end

```
iv.insert(iv.end(), 3);
```

Erase Element: erase()

- `v.erase(p)`
  - Removes element referred to by iterator `p`
  - Returns an iterator referring to the element **after** the one deleted, or an **off-the-end** iterator if p referred to the last element
  - p cannot be an **off-the-end** iterator
  - Example use: find an element and erase it

Reference

- **C++ Primer (4th Edision)**, by *Stanley Lippman, Josee Lajoie, and Barbara Moo*, Addison Wesley Publishing (2005)
  - Chapter 3.3 Library vector Type
  - Chapter 9 Sequential Containers
  - Chapter 14.8 Call Operator and Function Objects