C++

Function objects

Overview

Function pointers

Function objects

Introduction

Container: object holding other objects.

Provides limited set of methods for its manipulation.

To be useful we need more functions to: manipulate its size, iterate, copy, sort and search for elements.

The standard library provides algorithms to perform these basic needs.

Customization of the behavior of these algorithms is done through function objects.

Function pointers

- Function pointers are inherited from C
- A function pointer is an address in memory to executable code
- Example from the C library: qsort
  - qsort(void* base, size_t nmemb, size_t size, int (*compare) (const void*, const void*));
  - 4^{th} argument is of type int (*) (const void*, const void*) and is a function pointer

Example

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

int min (int a, int b) {
  return a < b ? a : b;
}
int max (int a, int b) {
  return a > b ? a : b;
}

int main() {
  bool maxQ = true;
  int (*f) (int a, int b);
  if (maxQ) f = min;
  else f = max;
  cout << f(1, 3) << endl;
}
```

f is the name of a var of type function pointer. It points to a fun that take two ints and return an int

Assign the address of either **min** or **max** to f.

Example 2: find_if

- **find()** and **find_if()** are functions from the standard library (header <algorithm>)
- **find()** returns the first iterator (in a range) matching an exact value (passed as argument).
- **find_if()** returns the first iterator (in a range) matching a predicate (passed as argument).

Example2: find_if

```
bool lt_5 (int a) { return a < 5; }

int main () {
  vector<int> v;
  v.push_back(11); v.push_back(9);
  v.push_back(7); v.push_back(5);
  v.push_back(2); v.push_back(8);

  vector<int>::iterator res;
  res = find_if(v.begin(), v.end(), lt_5);

  if (res != v.end()) cout << *res <<
endl;
}
```

Define a function "lt_5"

Returns the first iterator (it) in [v.begin(), v.end()) such that lt_5(*it) == true (i.e. *it < 5)

Output:
2

Function object

- A **function object** or **functor** is a generalization of the concept of function pointer
- Any object overloading **operator()** can be "called" as if it were a function
- Example:

```
class PlusA {
public:
  int _a;
  PlusA () : _a(1) {}
  PlusA (int a) : _a(a) {}
  int operator() (int b) { return _a + b; }
};
```

```
// …
PlusA
plus_a;
plus_a(5);
```

Function object

```
class PlusA {
public:
  int _a;
  PlusA () : _a(1) {}
  PlusA (int a) : _a(a) {}
  int operator() (int b) { return _a + b; }
};
```

```
// …
PlusA
plus_a;
plus_a(5);
```

- In the previous example:
  - PlusA is a class
  - PlusA plus_a creates an object; it is constructed by the default constructor which sets the internal parameter _a to 1
  - Operator() has been overloaded so plus_a(value) is a function that returns value + _a

Function object

- A function object (like any object) has a state.
- In the previous example, the state is defined by the member data **_a** (=1 in the example).
- Having a state overcomes a limitation of function pointers.

Example: find_if()

- We rewrite the previous example using find_if() with a function object

```
class LT {
private:
  int _a;
public:
  LT(int a) : _a(a) {}
  bool operator() (int b) { return b < _a; }
};
```

```
int main () {
  vector<int> v;
  v.push_back(11); v.push_back(9);
  v.push_back(7); v.push_back(5);
  v.push_back(2); v.push_back(8);

  vector<int>::iterator res;
  LT lt_5(5);
  res = find_if(v.begin(), v.end(), lt_5);

  if (res != v.end()) cout << *res <<
endl;
}
```

Example: find_if

- **find_if** examines each element of the container (by using an iterator **it**)
- For each element, it will check the predicate, by calling the functor lt_5 (3rd argument) with argument *it:
  - lt_5(*it); // lt_5.operator()(*it);

Function object bases

The standard library provide base classes to help writing function objects: unary_function and binary_function.

They are defined in header <functional> and are in the namespace std.

The goal of these functions is to provide standard names for the argument and return types.
For unary_function: argument_type and result_type
For binary_function: first_argument_type, second_argument_type and result_type.

Function objects in the standard library

In addition to the base classes, the standard library provides some function objects.

They are defined in header <functional> (namespace std).

They can be classified in:
predicate: function object that returns a bool (equal_to, not_equal_to, ...)
arithmetic: function object providing numeric operation (plus, minus, ...)
binder, adapter and negater: for composing function objects

Predicate

Name	Arity	Description
equal_to	binary	arg1 == arg2
not_equal_to	binary	arg1 != arg2
less	binary	arg1 < arg2
greater	binary	arg1 > arg2
less_equal	binary	arg1 <= arg2
greater_equal	binary	arg1 >= arg2
logical_and	binary	arg1 && arg2
logical_or	binary	arg1 || arg2
logical_not	unary	!arg

Arithmetic

Name	Arity	Description
plus	binary	arg1 + arg2
minus	binary	arg1 - arg2
multiplies	binary	arg1 * arg2
divides	binary	arg1 / arg2
modulus	binary	arg1 % arg2
negate	unary	-arg

Binder, adapter, negater

Binder allows to use a two-argument function as a one-argument function by binding one argument to a value

Adapter
Member function adapter allows a member function of a user defined class to be used as an argument to algorithms

Pointer function adapter allows a pointer function to be used as an argument to algorithms
negater negates a predicate

Summary

- An object that can be called like a function is a **function object** (or **functor**)
- A function object is a generalization of a function pointer:
  - Unlike function pointer, it can have data members and an internal state
- A function object must have the method **operator()** overloaded
- The standard library provides base classes and helpers