COMP6771
Advanced C++ Programming

Week 4
Part Three: Function Objects and Lamda Functions

2016

www.cse.unsw.edu.au/~cs6771

Converting Class Types to bool

- Convenient to use a class object as `bool` to test its state:

```
1  int v;
2  while (std::cin >> v) {
3    // ... do something with v
4  }
```

- A conversion operator in `istream` and `ostream` is defined:

```
operator bool () const { /* ... */ }
```

- The loop above is compiled into:

```
1  int v;
2  while ((cin >> v).operator bool()) {
3    // ... do something with v
4  }
```

The Call Operator ()

- Must be defined as a member function
- Can be overloaded in a class to create a so-called function object or functor, i.e., an object that can act as a function.

```
1   struct AbsInt {
2     int operator() (const int val) const {
3       return (val < 0) ? -val : val;
4     }
5   };
6
7  User code:
8
9   AbsInt absInt;
10  std::cout << absInt(-10) << std::endl;
```

- Can carry (and change) its own data values
- The only operator where the number of arguments is not fixed
- Used extensively by various STL algorithms (Chapter 10)

Structure of STL Algorithms Revisited

- Most STL algorithms take one of these forms:

```
1 std::alg(beg, end, other args)
2 std::alg(beg, end, dest, other args)
3 std::alg(beg, end, beg2, other args)
4 std::alg(beg, end, beg2, end2, other args)
```

- Some algorithms that take an *n*-ary predicate (or callable object) calls the callable object on the elements in the input range, where *n* is typically 1 (unary) or 2 (binary)
- Some use overloading to pass a predicate:

```
1 std::unique(beg, end);       // use operator== to compare
2 std::unique(beg, end, comp); // use binary predicate comp
```

- Some provide `_if` versions (to avoid ambiguity)

```
1 std::find(beg, end, val);  // find the 1st of val
2 std::find_if(beg, end, pred);
3      // find the 1st for which the unary pred is true
```

Callable Objects

- Five kinds of callable objects:
  - Functions
  - Pointers to functions
  - Objects of a class that overloads ()
  - Objects created by bind
  - Objects created by lamdba expressions
- The algorithmic sketch of `find_if`:

```
1 std::find_if(InputIterator __first, InputIterator __last,
2         Predicate __pred) {
3   while (__first != __last && !__pred(*__first))
4     ++__first;
5   return __first;
6 }
```

call the callable `__pred` on each element in the input range

Problem

- Consider the following vector:

```
1 std::vector<int> vec{ 20, 2, 4, 3, 8, 10, 15, 1};
```

- Problem: Write a program (or function) to find if any value in this vector is in the range 5 to 10.

Dumb Solution

- Problem: Write a program (or function) to find if any value in this vector is in the range 5 to 10.

```
1 bool checkRange() {
2   std::vector<int> vec{ 20, 2, 4, 3, 8, 10, 15, 1};
3
4   for (auto i : vec) {
5     if (i >= 5 && i <== 10) return true;
6   }
7   return false;
8 }
```

- Not very generic, e.g., what happens if we wanted values in the range 10 - 15?
- Can change function to take in parameters but still not the most effective approach.
- Fast solution: sort the vector and then go through the first `x` values, but what if we have a container we can't sort?
- Can we use STL algorithms to help us?

A Function as a Callable

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  // returns true if value is in range 5 to 10
6  bool range5to10(int val) {
7    return (5 <= val && val <= 10);
8  };
9
10 int main() {
11   std::vector<int> vec{ 20, 2, 4, 3, 8, 10, 15, 1};
12
13   // see if any value in the vector is in the range 5-10
14   auto presult = std::find_if(vec.begin(),vec.end(),range5to10);
15
16   if (presult == vec.end() )
17      std::cout << "[5, 10] not found" << std::endl;
18   else
19      std::cout << "[5, 10] found" << std::endl;
20 }
```

A Function Pointer as a Callable

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  bool range5to10(int val) {
6    return (5 <= val && val <= 10);
7  };
8
9  bool (*pf)(int) = range5to10;
10
11 int main() {
12   std::vector<int> vec{ 20, 2, 4, 3, 8, 10, 15, 1};
13
14   auto presult = std::find_if(vec.begin(), vec.end(), pf);
15
16   if (presult == vec.end() )
17      std::cout << "[5, 10] not found" << std::endl;
18   else
19      std::cout << "[5, 10] found" << std::endl;
20 }
```

A Function Object as a Callable

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  class range {
6  public:
7    range(int l, int u) : lb(l), ub(u) { }
8    bool operator() (int val) {
9       return (lb <= val && val <= ub);
10   }
11 private:
12   int lb, ub;
13 };
14
15 int main() {
16   std::vector<int> vec{ 20, 2, 4, 3, 8, 10, 15, 1};
17
18   auto presult = std::find_if(vec.begin(), vec.end(), range{5,10});
19
20   if (presult == vec.end() )
21      std::cout << "[5, 10] not found" << std::endl;
22   else
23      std::cout << "[5, 10] found" << std::endl;
24 }
```

More powerful than functions and pointers to functions because function objects contain state (i.e., the ranges here)

Function Objects vs. Functions

Consider the function GT6:

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  bool GT6(const std::string &s) { return s.size() >= 6; }
6
7  int main() {
8    std::vector<std::string> words{"function","objects","are","fun"};
9
10   std::cout << std::count_if(words.begin(), words.end(), GT6)
11        << " words have 6 characters or longer." << std::endl;
12 }
```

Output: `2 words have 6 characters or longer.`

Function Objects vs. Functions

- What if we want to find how many words are longer than x?
- We could define a couple of similar functions hardwired to the number x. But it's more flexible to use a function object.

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  class GTx {
6  public:
7    GTx(size_t val = 0): bound{val} {}
8    bool operator()(const std::string &s) { return s.size() >= bound; }
9  private:
10   std::string::size_type bound;
11 };
12
13 int main() {
14   std::vector<std::string> words{"function","objects","are","fun"};
15
16   std::cout << std::count_if(words.begin(), words.end(), GTx{6})
17   << " words have 6 characters or longer; "
18   << std::count_if(words.begin(), words.end(), GTx{3})
19   << " words have 3 characters or longer." << std::endl;
20 }
```

Output: `2 words have 6 characters or longer;`
`4 words have 3 characters or longer.`

Function Objects vs. Functions

We could now use a loop creating function objects to count the number of words with lengths greater than 3 through 9:

```
1 for (size_t i = 3; i != 10; ++i)
2  std::cout << std::count_if(words.begin(), words.end(), GTx{i})
3     << " words " << i
4     << " characters or longer." << std::endl;
```

Output:
```
4 words have 3 characters or longer.
2 words have 4 characters or longer.
2 words have 5 characters or longer.
2 words have 6 characters or longer.
2 words have 7 characters or longer.
1 words have 8 characters or longer.
0 words have 9 characters or longer.
```

Defining a Template Function-Object Class

- A template class (examined later)

```
1 template <class T, T lb, T ub>
2 struct range {
3   bool operator() (T val) {
4     return (lb <= val && val <= ub);
5   }
6 };
```

- One compiler-instantiated class:

```
1 struct range<int, 5, 10> {
2   bool operator() (int val) {
3      return (lb <= val && val <= ub);
4   }
5 };
```

- Another compiler-instantiated class:

```
1 struct range<std::string, "abstract", "concrete"> {
2   bool operator() (std::string val) {
3      return (lb <= val && val <= ub);
4   }
5 };
```

A Templated Function Object as a Callable

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  template <class T, T lb, T ub>
6  struct range {
7    bool operator() (T val) {
8      return (lb <= val && val <= ub);
9    }
10 };
11
12 int main() {
13   std::vector<int> vec{ 20, 2, 4, 3, 8, 10, 15, 1};
14
15   auto presult = find_if(vec.begin(), vec.end(),
16                                  range<int, 5, 10>{});
17
18   if (presult == vec.end() )
19      std::cout << "[5, 10] not found" << std::endl;
20   else
21      std::cout << "[5, 10] found" << std::endl;
22 }
```

The Library `bind` Function

- A general-purpose function adaptor
- Takes a callable object (function, fp, functor) and generates a new one that adapts the parameter list of the original object

```
1 auto newCallable = std::bind(Callable, arg_list);
```

- Example:

```
1  #include <iostream>
2  #include <functional>
3
4  void f(int a, int b, int c) {
5    std::cout << a << " " << b << " " << c << std::endl;
6  }
7
8  int main() {
9     auto g1 = std::bind(f, std::placeholders::_1, std::placeholders::_2, 100);
10    g1(1, 2); // prints 1 2 100
11    auto g2 = std::bind(f, std::placeholders::_2, std::placeholders::_1, 100);
12    g2(1, 2); //prints 2 1 100
13 }
```

_n is the n-th parameter of the new callable g

A Bind Object as a Callable

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <functional>
5
6  class range {
7  public:
8    bool operator() (int lb, int ub, int val) {
9       return (lb <= val && val <= ub);
10   }
11 };
12
13 int main() {
14   std::vector<int> vec 20, 2, 4, 3, 8, 10, 15, 1;
15
16   auto presult = std::find_if(vec.begin(), vec.end(),
17                               std::bind(range{},5,10,std::placeholders::_1));
18
19   if (presult == vec.end() )
20      std::cout << "[5, 10] not found" << std::endl;
21   else
22      std::cout << "[5, 10] found" << std::endl;
23 }
```

This is how it works (conceptually):

```
1   b = std::bind(range{}, 5, 10);
2   auto presult = find_if(vec.begin(), vec.end(), b);
3   // for each element x in the range, call b(x), which calls
4   // range().operator()(5,10,x), where x is bound to _1
```

bind1st and bind2nd

- bind is a generalization of bind1st and bind2nd
- Don't need to worry about understanding bind1st and bind2nd but need to understand the idea of bind.

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  int main() {
6    std::vector<int> vec{ 20, 2, 4, 3, 8, 10, 15, 1};
7
8    auto c1 = std::count_if(v.begin(), v.end(),
9                    std::bind1st(std::greater<int>{}, 10));
10   // SAME AS
11   auto c1a = std::count_if(v.begin(), v.end(),
12                   std::bind(std::greater<int>{}, 10, std::placeholders::_1));
13
14   auto c2 = std::count_if(v.begin(), v.end(),
15                   std::bind2nd(std::greater<int>{}, 10));
16   // SAME AS
17   auto c2a = std::count_if(v.begin(), v.end(),
18                   std::bind(std::greater<int>{}, std::placeholders::_1, 10));
19 }
```

Library Function Objects (Table 14.2)

The library header functional provides a convenient set of arithmetic, relational and logical function-object template classes that may be used with the STL algorithms.

- negate<T>(arg)
- plus<T>(arg1, arg2)
- minus<T>(arg1, arg2)
- multiplies<T>(arg1, arg2)
- divides<T>(arg1, arg2)
- modulus<T>(arg1, arg2)
- equal_to<T>(arg1, arg2)
- not_equal_to<T>(arg1, arg2)
- less<T>(arg1, arg2)
- less_equal<T>(arg1, arg2)
- greater<T>(arg1, arg2)
- greater_equal<T>(arg1, arg2)
- logical_not<T>(arg)
- logical_and<T>(arg1, arg2)
- logical_or<T>(arg1, arg2)

Function Object Adaptors

In addition, some function adaptors are provided that further increase the flexibility of the function objects. They are functions which convert a function object to another function object.

These functions construct a function object that has the opposite behaviour of the unary/binary function object op:

- `not1(op)`, negates unary op
- `not2(op)`, negates binary op

For example, consider:

```
1 int values[] = {1,2,3,4,5};
2 int cx = std::count_if (values, values+5, not1(IsOdd()));
3 std::cout << cx << " elements with even values." << std::endl;
```

Output: `2 elements with even values.`

A Lambda as a Callable

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  int main() {
6    std::vector<int> vec{ 20, 2, 4, 3, 8, 10, 15, 1};
7
8    auto presult = std::find_if(vec.begin(), vec.end(),
9      [] (int val) { return 5 <= val && val <= 10; }
10   );
11
12   if (presult == vec.end() )
13      std::cout << "[5, 10] not found" << std::endl;
14   else
15      std::cout << "[5, 10] found" << std::endl;
16 }
```

- The compiler will generate an unnamed function object from an unnamed class that looks like something similar to:

```
1  class __range {
2  public:
3    bool operator()(int val) { return (5 <= val && val <= 10); }
4  };
```

- The call `find_if` becomes:

```
1  auto presult = std::find_if(vec.begin(), vec.end(), __range(5,10));
```

Lambda Expressions

- Can be considered as an unnamed, inline function:

```
1 [capture list] (parameter list) -> return type {
2   function body
3 }
```

  - `capture list`: a list of local variables available in the enclosing function
  - `parameter list` and `return type` can be omitted
  - Can store a lamda function into a named type:

- Example:

```
1   auto f = [ ] { return 10; }
2
3   std::cout << f() << std::endl; // prints 10
```

Closure: http://en.wikipedia.org/wiki/Closure_(computer_science)

Passing Arguments to a Lambda

```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4
5  bool isShorter(const std::string &s1, const std::string &s2) {
6      return s1.size() < s2.size();
7  }
8
9  int main() {
10
11     std::vector<std::string> words {"the", "quick", "red", "fox",
12         "jumps", "over", "the", "slow", "red", "turtle"};
13
14     std::stable_sort(words.begin(), words.end(),
15             [](const std::string &s1, const std::string &s2)
16               { return s1.size() < s2.size();});
17     // SAME AS
18     std::stable_sort(words.begin(), words.end(), isShorter);
19 }
```

Lambda Captures

- Capture by value

```
1 int main() {
2   int v1 = 10;
3   auto f = [v1] { return v1; };
4   v1 = 0;
5   auto j = f();  // j is 10;
6 }
```

f is stored a copy of v1 when it was created in line 3

- Capture by reference

```
1 int main() {
2   int v1 = 10;
3   auto f = [&v1] { return v1; };
4   v1 = 0;
5   auto j = f();  // j is 0;
6 }
```

  - f refers to v1; it doesn't store it
  - must make sure that the reference still exists when if is called

Capture by Value and Reference

```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4
5  void biggies(std::vector<std::string> &words,
6          std::vector<std::string>::size_type sz,
7          std::ostream &os = std::cout, char c = ' ') {
8
9      std::stable_sort(words.begin(), words.end(),
10             [](const std::string &a, const std::string &b)
11               { return a.size() < b.size();});
12
13     auto wc = std::find_if(words.begin(), words.end(),
14                 [sz](const std::string &a)
15                 { return a.size() >= sz; });
16
17     auto count = words.end() - wc;
18
19     std::for_each(wc, words.end(),
20        [&os, c](const std::string &s){os << s << c;});
21 }
22
23 int main() {
24     std::vector<std::string> words {"the", "quick", "red", "fox",
25          "jumps", "over", "the", "slow", "red", "turtle"};
26     biggies(words, 5);
27 }
```

An Equivalent Version Using a Function Object

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4
5  bool check_size(const std::string &s, std::string::size_type sz) {
6    return s.size() >= sz;
7  }
8
9  void biggies(std::vector<std::string> &words, std::vector<std::string>::size_type sz) {
10     std::stable_sort(words.begin(), words.end(),
11             [](const std::string &s1, const std::string &s2)
12               { return s1.size() < s2.size();} );
13
14     auto wc = std::find_if(words.begin(), words.end(),
15                 std::bind(check_size, std::placeholders::_1, sz));
16
17     auto count = words.end() - wc;
18     std::for_each(wc, words.end(),
19        [](const std::string &s) { std::cout << s << " "; });
20 }
21
22 int main() {
23     std::vector<std::string> words{"the", "quick", "red", "fox",
24       "jumps", "over", "the", "slow", "red", "turtle"};
25     biggies(words, 5);
26 }
```

Implicit Captures

```
1    // os implicitly captured by reference
2    std::for_each(wc, words.end(),
3       [&, c](const string &s){os << s << c;});
4
5    // c implicitly captured by value
6    std::for_each(wc, words.end(),
7       [=, &os](const string &s){os << s << c;});
```

- The first is = or &
- If it is =, the explicitly captured variables must be captured by reference
- If it is &, the explicitly captured variables must be captured by value

Return Type for Lambdas

```
1
2  int main() {
3    std::vector<int> vec{ 20, 2, 4, 3, 8, 10, 15, 1};
4    std::transform(v.begin(), v.end(), v.begin(),
5    [] (int i) { return i < 0 ? -i : i; });
6    // the return type deduced to be int
7
8    std::transform(v.begin(), v.end(), v.begin(),
9     [] (int i) { if (i < 0) return -i; else return i; });
10
11   // the return type deduced to be int
12
13   std::transform(v.begin(), v.end(), v.begin(),
14   [] (int i) -> int { if (i < 0) return -i; else return i + 10.0; });
15 }
```

- The last example is a "trailing return type".
- Must specify the return type when it cannot be reduced
- Typically deduced when there is a single return statement

Readings

To complement the lecture, please read:

- Chapter 14, on operator overloading
- §14.9.2 – 14.9.3 concerning resolution details which were not covered in the lecture
- take a look at the relevant topics in the C++ FAQ