Lecture #19 | Function objects

SE271 Object-oriented Programming (2017)

Prof. Min-gyu Cho

Today’s topic

- Function object and lambda expression

Revisit: Standard Container Summary

Type	Container	Internal data structure
Sequential	`vector<T,A>`	A variable-size vector
	`list<T,A>`	A doubly-linked list
	`forward_list<T,A>`	A singly-linked list
	`deque<T,A>`	A double-ended queue
	`array<T,A>`	A fixed-size array
Associative	`set<T,C,A>`	A set (a map with just a key and no value)
	`multiset<T,C,A>`	A set in which a value can occur many times
	`map<K,V,C,A>`	An associative array
	`multimap<K,V,C,A>`	A map in which a key can occur many times
Unordered	`unordered_map<K,V,H,E,A>`	A map using a hashed lookup
	`unordered_multimap<K,V,H,E,A>`	A multimap using a hashed lookup
	`unordered_set<T,H,E,A>`	A set using a hashed lookup
	`unordered_multiset<T,H,E,A>`	A multiset using a hashed lookup

* T: type, K: key, V: value, A: allocator, C: comparison, H: hash, E: equality

Revisit: selected algorithms

`p=find(b,e,x)`	p is the first p in [b:e) so that *p==x
`p=find_if(b,e,f)`	p is the first p in [b:e) so that f(*p)==true
`n=count(b,e,x)`	n is the number of elements *q in [b:e) so that *q==x
`n=count_if(b,e,f)`	n is the number of elements *q in [b:e) so that f(*q,x)
`replace(b,e,v,v2)`	Replace elements *q in [b:e) so that *q==v by v2
`replace_if(b,e,f,v2)`	Replace elements *q in [b:e) so that f(*q) by v2
`p=copy(b,e,out)`	Copy [b:e) to [out:p)
`p=copy_if(b,e,out,f)`	Copy elements *q from [b:e) so that f(*q) to [out:p)
`p=move(b,e,out)`	Move [b:e) to [out:p)
`p=unique_copy(b,e,out)`	Copy [b:e) to [out:p); don't copy adjacent duplicates
`sort(b,e)`	Sort elements of [b:e) using < as the sorting criterion
`sort(b,e,f)`	Sort elements of [b:e) using f as the sorting criterion
`(p1,p2)=equal_range(b,e,v)`	[p1:p2) is the subsequence of the sorted sequence [b:e) with the value v; basically a binary search for v
`p=merge(b,e,b2,e2,out)`	Merge two sorted sequences [b:e) and [b2:e2) into [out:p)

`less` function object

- A *function object* (or *functor*): an object that can be called like functions
  - Many standard algorithms and containers require functor as arguments
  - Functor can be defined by overloading `operator()()` function
  - Inlined all the time
- `less` function object (defined in `<functional>`) is a default function object for comparison used by many standard algorithms and containers
- Function definition of less

```
template <class T = void> struct less {
    bool operator()(const T& x, const T& y) const;
};
```

- Return values: `true` if `lhs < rhs`; otherwise, `false`

Example: defining `less` for user-defined type

```
class Player {
private:
    int id;
    string name;
public:
    Player(int id_, string name_) : id{id_}, name{name_} {}
    string& get_name() { return name; }
    friend less<Player>;
};

using PlayerNote = pair<Player, string>;

template<> struct less<Player> {
    bool operator()(const Player& lhs, const Player& rhs) const {
        return lhs.id < rhs.id;
    }
};
```

Example: defining `less` for user-defined type (cont.)

```
int main() {
    map<Player, string> notes;
    Player p1 {19900905, "Kim Yuna"};
    Player p2 {19890927, "Park Tae-hwan"};
    Player p3 {19800209, "Kim Dong-sung"};

    notes.insert(PlayerNote{p1, "Figure Skater; 2010 Olympic champion"});
    notes.insert(PlayerNote{p2, "Swimmer; 2008 Olympic champion"});
    notes.insert(PlayerNote{p3, "Short track speed skater; 1998 Olympic champion"});

    for (PlayerNote p: notes) {
        Player& player = p.first;
        string summary = p.second;
        cout << player.get_name() << ": " << summary << endl;
    }
}
```

Lambda expression

- A *lambda expression* (or a *lambda function*, a *lambda*) is a simplified notation for defining and using an anonymous function object, used for
  - Passing an operation as an argument for an algorithm
  - Providing *callbacks* (often in GUI or network programming)
- Example: `[capture_list] (parameter_list) -> return_type { body}`
  - *capture list*: local names used in the body, and calling convention (call by value or reference); delimited by [] (can be empty)
  - *parameter list*: specify what arguments the lambda expression requires, delimited by ()
  - (optional) `mutable` specifier:
  - (optional) `noexcept` specifier
  - (optional) *return type declaration*: given as `-> type`
  - *body*: the code to be executed, delimited by {}

Example: lambda expression with function object

```
// lambda_expression_hello_world.cpp
int main() {
    [] {cout << "Hello, World!\n"; }();
}

// function_object_hello_world.cpp
struct Hello {
    void operator()() const {
        cout << "Hello, World!\n";
    }
};

int main() {
    Hello hello;
    hello();
}
```

Example: lambda expression with function object

```
// lambda_expression_abs_comparison.cpp
int main() {
    vector<int> v{-3, 3, 4, 0, -2, -1, 2, 1, -4};
    sort(v.begin(), v.end(),
        [](int x, int y) {return abs(x) < abs(y);});
    for (auto x : v) cout << x << '\n';
}
// function_object_abs_comparison.cpp
struct abs_less {
    bool operator()(int x, int y) const {return abs(x) < abs(y);}
};
int main() {
    vector<int> v{-3, 3, 4, 0, -2, -1, 2, 1, -4};
    sort(v.begin(), v.end(), abs_less());
    for (auto x : v) cout << x << '\n';
}
```

Example: lambda expression with transform()

```
int main()
{
    for (int m = 2; m < 5; ++m) {
        vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        transform(v.begin(), v.end(), v.begin(), [m](int x){return x % m;});
        cout << "Modula " << m << '\n';
        for (auto x : v) cout << x << '\n';
    }
}
```

Example: lambda expression with reference

```
int main() {
    int prod = 1;
    vector<int> v{2, 3, 4};
    for_each(v.begin(), v.end(),
        [&prod](int x) -> void {prod *= x;});
    cout << prod << '\n';
}
```

Reading list

- Lambda expression
  - http://www.drdobbs.com/cpp/lambdas-in-c11/240168241
  - http://www.cprogramming.com/c++11/c++11-lambda-closures.html

ANY QUESTIONS?