Function objects and lambda expressions

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Function objects and lambda expressions

Function objects and lambda expressions are two mechanisms in C++ that fulfil/replace the role of normal functions, for instance in articulation with STL algorithms.

Quick overview:

- **Function objects** are objects whose type defines `operator()`. We can use invoke operator `()` over function objects like we do for functions.
- **Lambda expressions**, as in other programming languages, allow for succinct "in-line" definitions of (anonymous) functions.

Function objects

Function objects, also called **functors**, are objects whose type provides at least one overload of `operator()`.

Example definition:

```
class interval {
private:
  int start_, end_;
public:
  interval (int start, int end)
    : start_(start), end_(end) {}
  bool operator () (int x) const {
    return x >= start_ && x <= end_;
  }
};
```

Function objects (cont.)

Function objects can be “invoked” like functions through `operator()`. If `o` is a function object `o(args)` corresponds to calling `o.operator()(args)`.

For `interval` as defined previously we may have for instance:

```
interval in_2_10(2,10);
cout << boolalpha
     << in_2_10(1) << ' '
     << in_2_10(7) << '\n';
```

Output:

```
false true
```

Function objects (cont.)

Functions objects are often useful to customise behavior in association to STL algorithms.

In the following example, the `interval` type of the previous example is used to set the matching predicate used with `std::count_if`.

```
vector<int> v { 1, 23, 5, 6, 32, 0, 45, 3 };
cout << count_if(v.begin(), v.end(), interval(20,40));
```

```
2
```

Function objects (cont.)

The advantage of using function objects is that their use is easily customisable, given that they have an associated state / internal configuration.

For instance we could (easily) reformulate the previous example for a generic interval:

```
vector<int> v { ... };
int a, b; cin >> a >> b;
cout << count_if(v.begin(), v.end(), interval(a, b));
```

In contrast, standard functions have code that is defined a priori. As such, the behavior of functions cannot be customised in straightforward manner.

Lambda expressions

A **lambda expression**, also known as **an anonymous function**, is an expression that defines a function without an associated name. The term lambda expression originates from the lambda calculus formal system by Alonzo Church.

Lambda expressions in C++ (but also other languages, e.g., Python or Java) are often convenient to define helper functions in succinct manner.

In its simplest form the general syntax is:

```
[ ] (arguments) -> return_type { body }
```

Objects representing lambda expressions are invoked like functions.

Lambda expressions (cont.)

A first example - a function that computes $x^2 + y$ given values x and y of type `int`:

```
auto f = [] (int x, int y) -> int { return x * x + y; };
cout << f(1, 1) << ' ' << f(1, 2) << ' ' << f(2, 2) << '\n';
```

Output:

```
2 3 6
```

Lambda expressions (cont.)

The return type can often be omitted for lambda expressions, as it normally can be inferred by the compiler.

In the previous example

```
auto f = [] (int x, int y) -> int { return x * x + y; };
```

can be simplified to

```
auto f = [] (int x, int y) { return x * x + y; };
```

Usually the type of a lambda expression object is also omitted - we use `auto` as above for `f`. The type can be stated if necessary using `std::function`:

```
std::function<int(int,int)> f =
  [] (int x, int y) { return x * x + y; };
```

Lambda expressions (cont.)

Like function objects, lambda expressions are often useful as arguments to STL algorithms. For instance, we can rewrite one the previous examples

```
count_if(v.begin(), v.end(),
       interval(20,40));
```

as

```
count_if(v.begin(), v.end(),
         [] (int x)
         { return x >= 20 && x <= 40; } );
```

Lambda expressions (cont.)

Lambda expressions sometimes require the use of variables declared in the enclosing function, e.g.

```
int a, b; cin >> a >> b;
...
count_if(v.begin(), v.end(),
         [] (int x)
          { return x >= a && x <= b; } );
```

but we get compilation errors from not stating how these variables are "captured" by the lambda expression body.

```
variable 'a' cannot be implicitly captured
in a lambda with
no capture-default specified
```

Lambda expressions - capture clause

In its simplest form

```
[ ] (arguments) return_type { body }
```

a lambda expression has an empty **capture clause** expressed by `[]`. To use variables in `body` that may be in context (besides arguments) we need to specify a non-empty capture clause that captures variables either **by value** or **by reference**.

A capture clause of the form `[=]` captures all variables **by value**. The lambda expression object captures the values of external variables it uses at the time of construction.

A capture clause of the form `[&]` captures all variables **by reference**. In this case, the lambda expression object captures references the external variables it uses.

It is possible to capture some variables by value and others by reference, e.g. `[v1, &v2]` captures `v1` by value and `v2` by reference.

Lambda expressions - capture clause (cont.)

For instance, we can express the previous example as

```
// a and b captured by value
[ = ] (int x)
     { return x >= a && x <= b; }
```

or

```
// a and b captured by reference
[ & ] (int x)
     { return x >= a && x <= b; }
```

or also

```
// a captured by value, b by reference
[ a, &b ] (int x)
     { return x >= a && x <= b; }
```

Lambda expressions - capture clause (cont.)

Let's understand the difference ...

```
int i = 123;
auto f = [=] (int x) { return x + i; };
cout << f(1) << '\n'; // -> 124
i = 321;
cout << f(1) << '\n';; // -> 124 again
```

The lambda expression object captures the values of `i` when it is defined. Subsequent updates to `i` do not alter value used in the lambda expression.

Lambda expressions - capture clause (cont.)

Let's understand the difference ...

```
int i = 123;
auto f = [&] (int x) { return x + i; };
cout << f(1) << '\n'; // -> 124
i = 321;
cout << f(1) << '\n';; // -> 322
```

The lambda expression object captures `i` by reference. Every evaluation of the lambda expression results in using the current value of `i`, not just the one `i` had when the lambda expression was defined.

Lambda expressions - capture clause (cont.)

Additionally, variables captured by reference can be changed in the body of a lambda expression. For instance

```
vector<int> v { 10, 20, 30 };
int i = 0;
// Use of std::for_each
for_each(v.begin(), v.end(),
         [ &i ] (int x) {
            cout << i << ": " << x << '\n';
            i++; // Increment i
          });
cout << "i=" << i << '\n'; // 3 at the end
```

yields the following output

```
0: 10
1: 20
2: 30
i=3
```