【C++】Day40

Class	C++
Date	@January 18, 2022
Material	
Series Number	
Summary	

【Ch10】Generic Algorithm

10.3.3 Lambda Captures and Returns

When we define a lambda, the compiler generates a new(unnamed) class type that corresponds to that lambda.

For now, what's useful to understand is that when we pass a lambda to a function, we are defining both a new type and an object of that type: The argument is an unnamed object of this compiler-generated class.

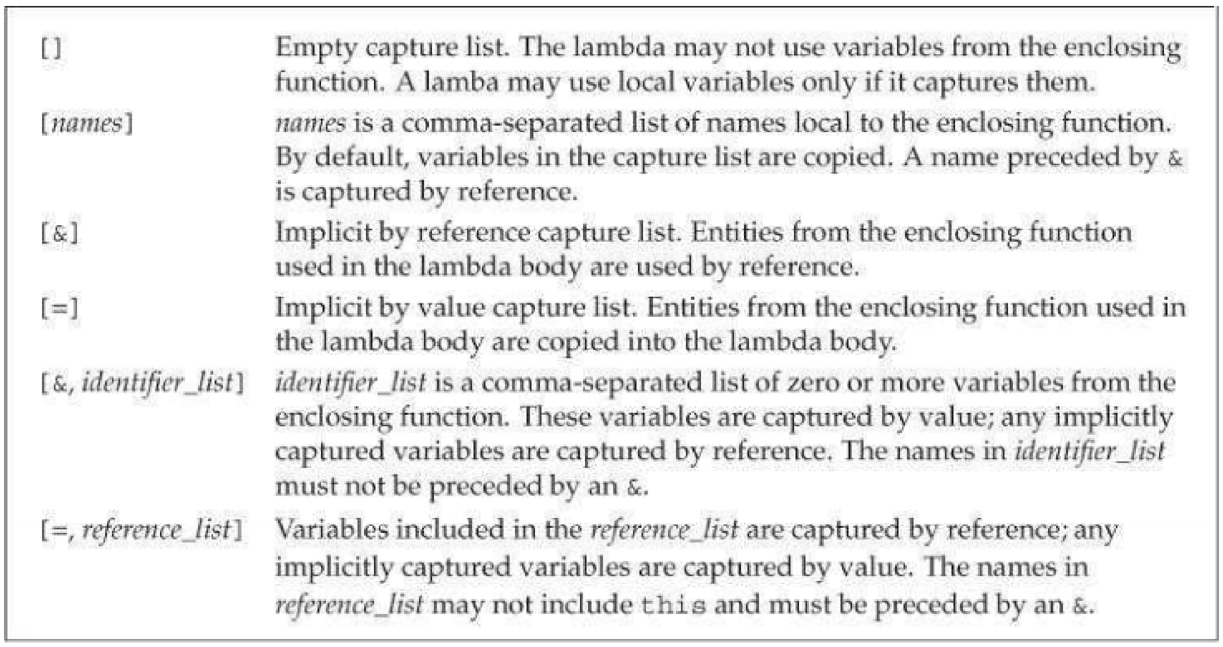
Similarly, when we use `auto` to define a variable initialized by a lambda, we are defining an object of the type generated from that lambda.

Capture by Value

Similar to parameter passing, we can capture varaibles by value or by reference.

The following table covers various ways we can form a capture list. As with a parameter passed by value, it must be possible to copy such variables.

Unlike parameters, the vaule of a captured variable is copied when the lambda is created, not when it is called.



[]	Empty capture list. The lambda may not use variables from the enclosing function. A lamba may use local variables only if it captures them.
[*names*]	*names* is a comma-separated list of names local to the enclosing function. By default, variables in the capture list are copied. A name preceded by & is captured by reference.
[&]	Implicit by reference capture list. Entities from the enclosing function used in the lambda body are used by reference.
[=]	Implicit by value capture list. Entities from the enclosing function used in the lambda body are copied into the lambda body.
[&, *identifier_list*]	*identifier_list* is a comma-separated list of zero or more variables from the enclosing function. These variables are captured by value; any implicitly captured variables are captured by reference. The names in *identifier_list* must not be preceded by an &.
[=, *reference_list*]	Variables included in the *reference_list* are captured by reference; any implicitly captured variables are captured by value. The names in *reference_list* may not include `this` and must be preceded by an &.

```
void func() {
  int val = 42;
  auto f = [val] () -> int { return val; };
  val = 0;
  std::cout << f(); //prints out 42,as val is copied when it is created.
}
```

Capture by Reference

We can also define lambdas that capture variables by reference. For example:

```
void func2() {
  size_t v1 = 42; //local variable
  //the object f2 contains a reference to v1
  auto f2 = [&v1] {return v1;};
  v1 = 0;
  std::cout << f2(); //print out 0, f2 refers to v1, does not store it
}
```

Warning: When we capture a variable by reference, we must ensure that the variable exists at the time that the lambda executes.

Implicit Captures

Rather than explicitly listing the variables we want to use from the enclosing function, we can let the compiler infer which variables we use from the code in the lambda's body.*(i.e the copmiler detects which variables we use and copy the value/refer to the value for us)*

To direct the compiler to infer the capture list, we use an `&` or `=` in the capture list. The `&` tells the copmiler to capture by reference, and the `=` says the values are captued by value.

For example, we can rewrite the lambda that we passed to `find_if` as

```
//sz implicitly captured by value
wc = find_if(words.begin(), words.end(), [=] (const string &s) -> bool { return s.size() >= sz});
```

If we want to capture som variables by value and others by reference, we can mix implicit and explicit captures:

```
void biggies(vector<string> &words, vector<string>::size_type sz, ostream &os = cout, char c = ' ') {
  for_each(words.begin(), words.end(), [&, c] (const string &s) { os << s << c});
}
```

When we mix implicit and explicit captures, the first item in the capture list must be an `&` or `=`.

Mutable Lambdas

By default, a lambda may not change the value of a variable that it copies by value. If we want to be able to change the value of a captured variable, we must follow the parameter list with the keyword `mutable`. Lambdas that are mutable may not omit the parameter list:

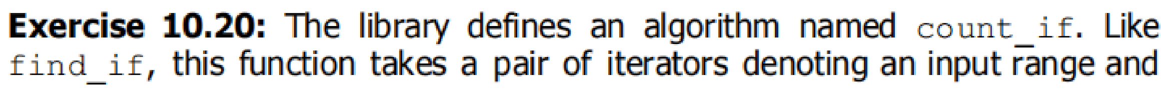
```
void fcn3() {
  size_t v1 = 42; //local variable
  auto f = [v1] () mutable { return ++v1; };
  v1 = 0;
  std::cout << f(); //print out 43
}
```

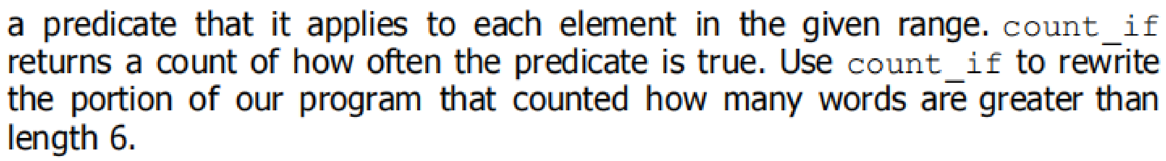
Specifying the Lambda Return Type

By default, if a lambda body contains any statements other than a return, that lambda is assumed to return void.

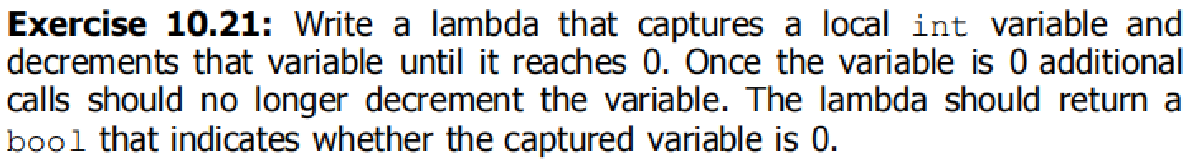
We have to expliticly declare the return type of our lambda expression if inside it has more than just one statemenet.

Exercise


Exercise 10.20: The library defines an algorithm named `count_if`. Like `find_if`, this function takes a pair of iterators denoting an input range and


a predicate that it applies to each element in the given range. `count_if` returns a count of how often the predicate is true. Use `count_if` to rewrite the portion of our program that counted how many words are greater than length 6.

See 10_20.cpp for code


Exercise 10.21: Write a lambda that captures a local `int` variable and decrements that variable until it reaches 0. Once the variable is 0 additional calls should no longer decrement the variable. The lambda should return a `bool` that indicates whether the captured variable is 0.

See10_21.cpp for code