day-9-lambda-notation

September 12, 2023

1 Day 9: Lambda notation

1.0.1 Topics

- lambda notation
- captureless lambdas
- capturing lambdas

1. Lambda notation A *lambda expression* is an object which is callable (just like a function or an object method) and is capable of capturing variables from the surrounding scope(s)

Formally, a lambda is a closure.

If you have programmed in Python before, the concept of a lambda in C++ is very much like Python's functions and lambda expressions.

The most basic lambda expression is one which does nothing:

```
auto f = [](){};
```

The object `f` is a lambda expression which captures no variables, accepts no arguments, does nothing when called and returns nothing. Pretty useless so far.

Here is a lambda which just returns an `int`:

```
auto g = [](){ return 42; };
```

Notice that you don't need to state the return type of this lambda - the compiler can infer the return type for us.

The `()` in the lambda expression defines the parameters of the lambda:

```
auto sum = [](int i, int j) { return i+j; };
```

Here, `sum` is a lambda which accepts two `int`s and returns their sum.

1.1. Basic lambda expressions

Complete the code below, defining lambdas which accept two integers and return their:

- sum
- product
- minimum
- maximum

Use these lambdas to operate on the vectors of values to find the total, product, min and max of all values in the vectors.

[ ]:
```cpp
//// lambda.cpp
#include <iostream>
#include <vector>

using std::cout;


int main() {

    std::vector<int> v1 = {1,2,3,4,5};
    std::vector<int> v2 = {42, 19, -45, 0, 16};

    auto sum = [](){}; // implement me!

    // loop over the elements of v1
    int total = 0;
    for (auto k: v1) {
        total = sum(total, k);
    }
    cout << total << "\n";

    return 0;
}
```

1.2. More lambda expressions

Write new versions of these lambda expressions which each accept a `std::vector<int>` and return the sum/product/min/max of the elements in the vector.

[ ]:
```cpp
//// lambdas-of-vectors.cpp
#include <iostream>
#include <vector>

using std::cout;

int main() {

    std::vector<int> v1 = {1,2,3,4,5};

    auto sum = [](std::vector<int>& v) {}; // implement me!

    cout << "The sum of elements of v1 is " << sum(v1) << "\n";

    return 0;
}
```

2. Captureless lambdas Lambdas with empty capture groups (the `[]` in the declaration) are implicitly convertible to function pointers of the same call signature. Consider this function:

```
void print_result(int (*f)()) {
    cout << "result is " << f() << "\n";
}
```

The function `print_result` accepts one parameter `f` which is a pointer to a function that has no parameters and itself returns an int. The function `print_result` returns nothing, but prints the result of calling `f`.

Below is an example using function pointers and non-capturing lambdas.

[ ]:
```
//// captureless.cpp
#include <iostream>

using std::cout;

void print_result(int (*f)()) {
    cout << "result is " << f() << "\n";
}

int f1() {
    return 19;
}

int main(void) {

    // use a function pointer
    print_result(f1);

    // use a captureless lambda
    auto f2 = [](){ return 42; };
    print_result(f2);

    return 0;
}
```

Write a function `aggregate` with this signature:

```
int aggregate(std::vector<int> v, int start, int (*combine)(int lhs, int rhs));
```

The arguments are a `std::vector<int>`, an `int` starting value, and a pointer to a function which accepts two `int`s and returns an `int`.

The function specified in the `combine` parameter shall combine it's two `int` parameters in some way e.g. their sum or product. The `aggregate` function should apply the `combine` parameter to every element in `v`, with the initial value `start`.

For example, given this function:

```
int sum(int lhs, int rhs) { return lhs + rhs; }
```

And the vector `std::vector<int> v = {1,2,3,4,5}`, the result of calling `aggregate(v, 0, sum)` should be 15.

Test your `aggregate` function with a few captureless lambdas which return the sum, product, min and max of a pair of integers.

3. Capturing lambdas The real power of lambdas lies in their ability to remember or ``capture'' values from the scope in which they are created. Lambdas can capture values either by reference or by value. This lambda captures everything by reference:

```
auto capture_by_ref = [&](){};
```

This lambda captures everything by value:

```
auto capture_by_val = [=](){};
```

The capture group (in `[]`) can contain a comma-separated list of variables to capture, each with their own method of capture e.g. the capture group `[&p, v]` captures the variable `p` by reference and the variable `v` by value.

Let's explore the difference with some exercises.

Write a program which uses capturing lambdas to print the length of a captured `std::vector<int>`, captured both by reference and by value. Your lambdas should only capture this vector, and should accept no parameters when called.

Which results do you expect when calling these lambdas? Reason about the results before executing the code.

[ ]:
```
//// capturing.cpp
#include <iostream>
#include <vector>

using std::cout;

int main() {

    std::vector<int> v = {1,2,3,4,5};

    auto by_value = [](){}; // capture v by value and print its length
    auto by_ref = [](){}; // capture v by reference and print its length

    return 0;
}
```

After calling these lambdas once, append some more values to the vector `v` and call both lambdas again. Reason about the results before executing the code. Are the results what you expected? Can you explain the results?