Lambda & std::function

terça-feira, 12 de novembro de 2024 07:13

It is a convenient way to define an anonymous function object or functor. It is convenient because we can define it locally where we want to call it or pass it to a function as an argument.

A callback function is a function that is passed as an argument to another function and is then called within that function.

C++ Lambda Expressions

This is how we define a lambda in C++:

```
auto plus_one = [](const int value)
{
    return value + 1;
};

assert(plus_one(2) == 3);
```

`plus_one` in this code is a functor under the hood. Now let's see what a functor is.

Lambda functions in C++ are essentially unnamed, inline functions that are typically used when you need a simple function temporarily. They're useful for concise code and are frequently used in algorithms and callbacks.

1. Basic Syntax

A lambda function in C++ has the following general syntax:

cpp Copy code

```
[capture](parameters) -> return_type {
    // function body
};
```

Here's a breakdown of each part:

- **Capture**: `[ ]` - This is where you specify which variables from the surrounding scope the lambda should capture. what should be defined within it
- **Parameters**: `( )` - These are the input parameters for the lambda, similar to any regular function.
- **Return Type**: `-> return_type` - You can specify the return type here. If omitted, C++ tries to deduce it automatically.
- **Function Body**: `{ }` - This contains the code that the lambda will execute.

2. Capturing Variables

The capture section lets the lambda access variables from its surrounding scope. You can capture variables in various ways:

- **By value** `[=]` - Captures all variables by value (a copy of the variables).
- **By reference** `[&]` - Captures all variables by reference (you work with the original variables).
- **Mixed capture** - You can specify individual variables and whether they're by value or reference:
  - `[x, &y]` - Captures `x` by value and `y` by reference.

Example:

cpp Copy code

```cpp
int a = 5;
int b = 10;
auto lambda = [a, &b](int x) -> int {
    b += x;  // Modifies 'b' since it's captured by reference
    return a + b + x;  // Accesses 'a' by value
};
```

3. Example of a Simple Lambda

Here's a lambda that adds two numbers:

cpp Copy code

```cpp
auto add = [](int x, int y) -> int {
    return x + y;
};
int result = add(2, 3);  // result is 5
```

5. Mutable Lambdas

By default, lambdas are `const`, meaning they can't modify captured variables if they're captured by value. To allow modification, use the `mutable` keyword.

cpp Copy code

```cpp
int value = 5;
auto increment = [value]() mutable {
    value++;  // This modification only affects the captured copy
    return value;
};
int result = increment();  // result is 6, but 'value' remains 5 in the outer scope
```

6. Lambda Expressions as Function Parameters

Lambdas can be passed directly to functions that accept function pointers or other callable types.

cpp

Copy code

```cpp
void operateOnVector(const std::vector<int>& vec, std::function<void(int)> func) {
    for (int val : vec) {
        func(val);
    }
}

std::vector<int> values = {1, 2, 3, 4};
operateOnVector(values, [](int x) {
    std::cout << x << " ";
});
```

Here is the lambda definition
↳ Define what it can use from the scope
↳ define the inputs
↳ Define the logic within it