Shahar Mike's Web Spot

Yet another geek's blog

Under the hood of lambdas and std::function

(1295 words)
Tue, Feb 23, 2016

Table of Contents

In this post we'll explore how lambdas behave in different aspects. Then we'll look into `std::function` and how it works.

What's a lambda?

Here's a quick recap if you have yet to use one of the most powerful features of C++11 – lambdas:

Lambdas are a fancy name for anonymous functions. Essentially they are an easy way to write functions (such as callbacks) in the logical place they should be in the code.

My favorite expression in C++ is `[](){}();`, which declares an empty lambda and immediately executes it. It is of course completely useless. Better examples are with STL, like:

```
std::sort(v.begin(), v.end(), [](int a, int b) { return a > b; });
```

This has the following advantages over C++98 alternatives: it is where the code would logically be (as opposed to defining a class/function somewhere outside this scope), and it does not pollute any namespace (although this could be easily be bypassed even in C++98).

Syntax

Lambdas have 3 parts:

1. Capture list – these are variables that are copied inside the lambda to be used in the code;
2. Argument list – these are the arguments that are passed to the lambda at execution time;
3. Code – well.. code.

Here's a simple example:

```
int i = 0, j = 1;
auto func = [i, &j](bool b, float f){ ++j; cout << i << ", " << b <<
func(true, 1.0f);
```

1. First line is simple - create 2 `int`s named `i` and `j`.
2. Second line defines a lambda that:
    - Captures `i` by value, `j` by reference,
    - Accepts 2 parameters: `bool b` and `float f`,
    - Prints `b` and `f` when invoked
3. Third line calls this lambda with `true` and `1.0f`

I find it useful to think of lambdas as classes:

- Captures are the data members:
    - The data members for `f` above are `i` and `j`;
    - The lambda can access these members inside it's code scope.
- When a lambda is created, a constructor copies the captured variables to the data members;
- It has an `operator()(...)` (for `f` the `...` is `bool, float`);
- It has a scope-lifetime and a destructor which frees members.

One last thing syntax-wise: you can also specify a default capture:

- `[&](){ i = 0; j = 0; }` is a lambda that captures `i` and `j` by reference. `[&]` means 'capture by-reference all variables that are in use in the function'

- `[=](){ cout << k; }` is a lambda that captures `k` by value. Similarly, `[=]` means 'capture by-value all variables that are in use in the function'
- You can also mix and match: `[&, i, j](){}` captures all variables by reference except for `i` and `j` which are captures by value. And of-course the opposite is also possible: `[=, &i, &j](){}`.

Capture by value vs by reference

Above we mentioned capturing a lambda by value vs by reference. What's the difference? Here's a simple code that will illustrate:

```
int i = 0;
auto foo = [i](){ cout << i << endl; };
auto bar = [&i](){ cout << i << endl; };
i = 10;
foo();
bar();
```

```
0
10
```

Lambda's type

One important thing to note is that a lambda is not a `std::function`. It is true that a lambda *can be assigned to* a `std::function`, but that is not its native type. We'll talk about what that means soon.

As a matter of fact, there is no standard type for lambdas. A lambda's type is implementation defined, and the only way to capture a lambda with no conversion is by using `auto`:

```
auto f2 = [](){};
```

However, if your capture list is empty you may convert your lambda to a C-style function pointer:

```
void (*foo)(bool, int);
foo = [](bool, int){};
```

Lambda's scope

All captured variables have the scope of the lambda:

```
#include <iostream>
#include <functional>
```

```
struct MyStruct {
        MyStruct() { std::cout << "Constructed" << std::endl; }
        MyStruct(MyStruct const&) { std::cout << "Copy-Constructed" <
        ~MyStruct() { std::cout << "Destructed" << std::endl; }
};

int main() {
        std::cout << "Creating MyStruct..." << std::endl;
        MyStruct ms;

        {
                std::cout << "Creating lambda..." << std::endl;
                auto f = [ms](){}; // note 'ms' is captured by-value
                std::cout << "Destroying lambda..." << std::endl;
        }

        std::cout << "Destroying MyStruct..." << std::endl;
}
```

Output:

```
Creating MyStruct...
Constructed
Creating lambda...
Copy-Constructed
Destroying lambda...
Destructed
Destroying MyStruct...
Destructed
```

`mutable` lambdas

lambda's `operator()` is const by-default, meaning it can't modify the variables it captured *by-value* (which are analogous to class members). To change this default add `mutable`:

```
int i = 1;
[&i](){ i = 1; }; // ok, 'i' is captured by-reference.
[i](){ i = 1; }; // ERROR: assignment of read-only variable 'i'.
[i]() mutable { i = 1; }; // ok.
```

This gets even more interesting when talking about copying lambdas. Key thing to remember - they behave like classes:

```
int i = 0;
auto x = [i]() mutable { cout << ++i << endl; }
x();
auto y = x;
x();
y();
```

```
1
2
2
```

Lambda's size

Because lambdas have captures, there's no single size for all lambdas. Example:

```
auto f1 = [](){};
cout << sizeof(f1) << endl;

std::array<char, 100> ar;
auto f2 = [&ar](){};
cout << sizeof(f2) << endl;

auto f3 = [ar](){};
cout << sizeof(f3) << endl;
```

Output (64-bit build):

```
1
8
100
```

Performance

Lambdas are also awesome when it comes to performance. Because they are objects rather than pointers they can be inlined very easily by the compiler, much like functors. This means that calling a lambda many times (such as with `std::sort` or `std::copy_if`) is much better than using a global function. This is one example of where C++ is actually faster than C.

`std::function`

`std::function` is a templated object that is used to store and call any *callable* type, such as functions, objects, lambdas and the result of `std::bind`.

Simple example

```
#include <iostream>
#include <functional>
using namespace std;

void global_f() {
        cout << "global_f()" << endl;
}

struct Functor {
        void operator()() { cout << "Functor" << endl; }
};

int main() {
        std::function<void()> f;
        cout << "sizeof(f) == " << sizeof(f) << endl;

        f = global_f;
        f();

        f = [](){ cout << "Lambda" << endl;};
        f();

        Functor functor;
        f = functor;
        f();
}
```

Output:

```
$ clang++ main.cpp -std=c++14 && ./a.out
sizeof(f) == 32
global_f()
Lambda
Functor
```

`std::function`'s Size

On clang++ the size of all `std::function`s (regardless of return value or parameters) is always 32 bytes. It uses what is called *small size optimization*, much like `std::string` does on many implementations. This basically means that for small objects `std::function` can keep them as part of its memory, but for bigger objects it defers to dynamic memory allocation. Here's an example on a 64-bit machine:

```
#include <iostream>
#include <functional>
#include <array>
#include <cstdlib> // for malloc() and free()
using namespace std;

// replace operator new and delete to log allocations
void* operator new(std::size_t n) {
        cout << "Allocating " << n << " bytes" << endl;
        return malloc(n);
}
void operator delete(void* p) throw() {
        free(p);
}

int main() {
        std::array<char, 16> arr1;
        auto lambda1 = [arr1](){};
        cout << "Assigning lambda1 of size " << sizeof(lambda1) << er
        std::function<void()> f1 = lambda1;

        std::array<char, 17> arr2;
        auto lambda2 = [arr2](){};
        cout << "Assigning lambda2 of size " << sizeof(lambda2) << er
        std::function<void()> f2 = lambda2;
}
```

```
$ clang++ main.cpp -std=c++14 && ./a.out
Assigning lambda1 of size 16
Assigning lambda2 of size 17
Allocating 17 bytes
```

17. That's the threshold beyond which `std::function` reverts to dynamic allocation (on clang). Note that the allocation is for the size of 17 bytes as the lambda object needs to be contiguous in memory.

That's it for my first post. I hope you enjoyed reading it as much as I enjoyed writing it. Please let me know if you have any suggestions, questions or comments!

C++ vtables - Part 1 - Basics >>

Comments

```
&& sizeof(_Functor) <= _M_max_size
&& __alignof__(_Functor) <= _M_max_align
&& (_M_max_align % __alignof__(_Functor) == 0));


typedef integral_constant<bool, __stored_locally=""> _Local_storage;
```

In which _Local_storage affects how std::function will be created.

see more

• Reply • Share ›

Aymen Schehaider • a year ago

liked the overloading allocation new :) refreshed me some technics

• Reply • Share ›

vitalyx • a year ago

Awesome, thank your for writing this!

• Reply • Share ›

Sir Percival • 2 years ago

Nice and informative article.
Topic little misleading though as this is pretty much an introduction and how to apply lambdas. I was looking for how lambdas are implemented.

• Reply • Share ›

Ivan Zhang • 2 years ago

I tried the last sample on Ubuntu 64bits with clang++-3.5.
The result is, it always allocate memory, even if I changed the code to :
std::array<char, 1=""> arr1;

• Reply • Share ›

shaharmike Mod ➚ Ivan Zhang • 2 years ago

I tried this with clang++ 3.5 and the threshold seems to be 24:
https://wandbox.org/permlin...
Maybe you're using an old libstdc++? It seems like, at least in older versions, there's no SOO there.

• Reply • Share ›

hkBattousai • 3 years ago • edited

Why do the all lambda examples have the return type of `void` here?

• Reply • Share ›

shaharmike Mod ➚ hkBattousai • 2 years ago

That's a very good observation. Admittedly I did not cover return values, sorry!

• Reply • Share ›

Jens Stimpfle • 4 years ago

Regarding "performance" and lambdas being "objects rather than pointers": Lambdas aren't magically better than plain function pointers. They are just a syntax for ad-hoc function

definition (the definition is "inline", but that's not the same as an inlined call). What a compiler needs to inline a function call is static knowledge what function will be called.

Reply • Share ›

shaharmike Mod ➚ Jens Stimpfle • 2 years ago

You are absolutely correct.

Reply • Share ›

Sakari Lehtonen • 4 years ago

Nice and well written, concise article. Thanks for writing and sharing this. Lambdas are neat! :)

Reply • Share ›

shaharmike Mod ➚ Sakari Lehtonen • 2 years ago

Neat indeed, thanks!

Reply • Share ›

Lichtso • 4 years ago

I think the way std::function works is also quiet interesting. As each lambda has its own class / type you can't pass them around or fit them into the same variable / parameter because of their type conflict. And another thing is that you can only get the data pointer but not the function pointer. But on the other hand, std::function as part of the std lib is written in C++ itself, so how does it work ... turns out it is using vtables. I've never seen a post on this topic and it would be nice, as yours is very similar.

Reply • Share ›

Johannes Dahlström ➚ Lichtso • 4 years ago

The technique is called type erasure - here's a nice three-part article explaining it:

https://akrzemi1.wordpress....

Reply • Share ›