C++ Lambdas Under The Hood

Introduction

C++11 introduced lambdas, which provide a syntactically lightweight way to define functions on-the-fly. They can also capture (or close over) variables from the surrounding scope, either by value or by reference. In this article, we investigate how lambdas differ, implementation-wise, from plain functions and functor classes (classes that implement `operator()`).

Source files, along with a makefile to generate disassemblies, are available here. All assembly code was generated by GCC version 6.2.1 with no optimizations and the `-g` flag enabled.

No Capturing

We first investigate callable things that do not capture any variables. C++ offers three alternatives: plain functions, functor classes, and lambdas. This listing demonstrates each approach for a simple operation:

```
1  int function (int a) {
2    return a + 3;
3  }
4
5  class Functor {
6    public:
7      int operator()(int a) {
8        return a + 3;
9      }
10 };
11
12 int main() {
13   auto lambda = [] (int a) { return a + 3; };
14
15   Functor functor;
16
17   volatile int y1 = function(5);
18   volatile int y2 = functor(5);
19   volatile int y3 = lambda(5);
20
21   return 0;
22 }
```

And below, we have the disassembly from each approach, along with the disassembler's best guess at what lines of the source file are associated with various parts of the assembly code. (Thus, certain lines may not appear or may appear multiple times; assembly does not necessarily map directly to C++!) I've pared the results down to just the relevant snippets.

Plain function:

```
102 00000000004004c6 <function(int)>:
103 int function (int a) {
104   4004c6: 55                    push   rbp
105   4004c7: 48 89 e5              mov    rbp,rsp
106   4004ca: 89 7d fc              mov    DWORD PTR [rbp-0x4],edi
107   return a + 3;
108   4004cd: 8b 45 fc              mov    eax,DWORD PTR [rbp-0x4]
```

```
109   4004d0: 83 c0 03              add    eax,0x3
110 }
111   4004d3: 5d                    pop    rbp
112   4004d4: c3                    ret
```

Here, the parameter `a` is passed in the register `edi`, and the return value is placed in the `eax` register. The assembly was not optimized at all, so lines 106 and 108 place the argument on the stack, then immediately retrieve it. (In Intel x86 assembly syntax, the use of `[]` works like a pointer dereference; `rbp` contains the "base pointer", a pointer to the top of the stack frame.) The stack frame is 4 bytes (hence the `rbp-0x4`) since there is only one 4-byte value stored on it. Line 109 actually performs the addition; the remainder of the lines consist of setting up and cleaning up various registers for the function call.

Functor:

```
162 000000000040052e <Functor::operator()(int)>:
163     int operator()(int a) {
164   40052e: 55                    push   rbp
165   40052f: 48 89 e5              mov    rbp,rsp
166   400532: 48 89 7d f8           mov    QWORD PTR [rbp-0x8],rdi
167   400536: 89 75 f4              mov    DWORD PTR [rbp-0xc],esi
168       return a + 3;
169   400539: 8b 45 f4              mov    eax,DWORD PTR [rbp-0xc]
170   40053c: 83 c0 03              add    eax,0x3
171     }
172   40053f: 5d                    pop    rbp
173   400540: c3                    ret
```

Lambda:

```
115 00000000004004d6 <main::{lambda(int)#1}::operator()(int) const>:
121   auto lambda = [] (int a) { return a + 3; };
122   4004d6: 55                    push   rbp
123   4004d7: 48 89 e5              mov    rbp,rsp
124   4004da: 48 89 7d f8           mov    QWORD PTR [rbp-0x8],rdi
125   4004de: 89 75 f4              mov    DWORD PTR [rbp-0xc],esi
126   4004e1: 8b 45 f4              mov    eax,DWORD PTR [rbp-0xc]
127   4004e4: 83 c0 03              add    eax,0x3
128   4004e7: 5d                    pop    rbp
129   4004e8: c3                    ret
```

The code generated from the functor class and the lambda are identical, but differ from the plain function in one way: there is a hidden first parameter passed in `rdi` (due to x86 weirdness, this is the same register as `edi`, but it holds 8 bytes instead of 4). It is unused in this function; we will see its purpose later on. The second parameter `a` is passed in `esi` (instead of `edi`, as with the plain function). Due to this hidden parameter, the stack frame is now 12 bytes; 4 for `a` and 8 for the hidden parameter.

Summary:

For plain functions that capture no variables, lambdas and functors behave the same. They differ from a plain C++ function only in that they take an additional hidden parameter, thus requiring an extra 8 bytes of stack space. (In addition, they require a single byte on `main()`'s stack that is related to the hidden parameter.)

Capture By Value

When capturing variables, we cannot use a standard C++ function, so we are left with two approaches: functor classes and lambdas. The code below implements a simple function using each technique.

```
 1 class Functor {
 2   public:
 3     Functor(const int x): m_x(x) {}
 4
 5     int operator()(int a) {
 6       return a + m_x;
 7     }
 8
 9   private:
10     int m_x;
11 };
12
13 int main() {
14   int x = 3;
15
16   auto lambda = [=] (int a) { return a + x; };
17   Functor functor(x);
18
19   volatile int y1 = functor(5);
20   volatile int y2 = lambda(5);
21
22   return 0;
23 }
```

Functor:

The functor now has two functions for us to examine, the constructor and the call operator:

```
157 0000000000400534 <Functor::Functor(int)>:
158     Functor(const int x): m_x(x) {}
159   400534: 55                    push   rbp
160   400535: 48 89 e5              mov    rbp,rsp
161   400538: 48 89 7d f8           mov    QWORD PTR [rbp-0x8],rdi
162   40053c: 89 75 f4              mov    DWORD PTR [rbp-0xc],esi
163   40053f: 48 8b 45 f8           mov    rax,QWORD PTR [rbp-0x8]
164   400543: 8b 55 f4              mov    edx,DWORD PTR [rbp-0xc]
165   400546: 89 10                 mov    DWORD PTR [rax],edx
166   400548: 90                    nop
167   400549: 5d                    pop    rbp
168   40054a: c3                    ret
```

The constructor's assembly is a very long way of saying "put the contents of `esi` into the memory location stored in `rdi`". To find out what is passed as the two arguments to this function, let's have a look at the code in `main()` that calls the constructor:

```
125   int x = 3;
126   4004e6: c7 45 fc 03 00 00 00  mov    DWORD PTR [rbp-0x4],0x3
...
130   Functor functor(x);
131   4004f3: 8b 55 fc              mov    edx,DWORD PTR [rbp-0x4]
132   4004f6: 48 8d 45 e0           lea    rax,[rbp-0x20]
133   4004fa: 89 d6                 mov    esi,edx
134   4004fc: 48 89 c7              mov    rdi,rax
135   4004ff: e8 30 00 00 00        call   400534 <Functor::Functor(int)>
```

So, we see (from line 126) that `x` is stored at `rbp-0x4`, and is (indirectly, via lines 131 and 133) copied into `esi`. Lines 132 and 134 place the address `rbp-0x20` into `rdi` (`lea` stands for "load effective address"). This address, naturally, is the `this` pointer for our functor object!

```
171 000000000040054c <Functor::operator()(int)>:
172     int operator()(int a) {
173   40054c: 55                    push   rbp
```

```
174   40054d: 48 89 e5             mov    rbp,rsp
175   400550: 48 89 7d f8          mov    QWORD PTR [rbp-0x8],rdi
176   400554: 89 75 f4             mov    DWORD PTR [rbp-0xc],esi
177       return a + m_x;
178   400557: 48 8b 45 f8          mov    rax,QWORD PTR [rbp-0x8]
179   40055b: 8b 10                mov    edx,DWORD PTR [rax]
180   40055d: 8b 45 f4             mov    eax,DWORD PTR [rbp-0xc]
181   400560: 01 d0                add    eax,edx
182     }
183   400562: 5d                   pop    rbp
184   400563: c3                   ret
```

The code for the actual function call is very similar to that of the plain function version. The major difference is on lines 178 and 179, which load the value of `m_x` stored by our constructor into `edx`.

Lambda:

```
102 00000000004004c6 <main::{lambda(int)#1}::operator()(int) const>:
108   auto lambda = [=] (int a) { return a + x; };
109   4004c6: 55                   push   rbp
110   4004c7: 48 89 e5             mov    rbp,rsp
111   4004ca: 48 89 7d f8          mov    QWORD PTR [rbp-0x8],rdi
112   4004ce: 89 75 f4             mov    DWORD PTR [rbp-0xc],esi
113   4004d1: 48 8b 45 f8          mov    rax,QWORD PTR [rbp-0x8]
114   4004d5: 8b 10                mov    edx,DWORD PTR [rax]
115   4004d7: 8b 45 f4             mov    eax,DWORD PTR [rbp-0xc]
116   4004da: 01 d0                add    eax,edx
117   4004dc: 5d                   pop    rbp
118   4004dd: c3                   ret
```

The code for the lambda function call is identical to that of our functor's `operator()`. No surprise there. But where is the constructor function? Let's look in `main()`:

```
120 00000000004004de <main>:
121 int main() {
122   4004de: 55                   push   rbp
123   4004df: 48 89 e5             mov    rbp,rsp
124   4004e2: 48 83 ec 30          sub    rsp,0x30
125   int x = 3;
126   4004e6: c7 45 fc 03 00 00 00  mov    DWORD PTR [rbp-0x4],0x3
127   auto lambda = [=] (int a) { return a + x; };
128   4004ed: 8b 45 fc             mov    eax,DWORD PTR [rbp-0x4]
129   4004f0: 89 45 f0             mov    DWORD PTR [rbp-0x10],eax
```

All the work of the functor's constructor is done in lines 128 and 129! The `this` pointer for the lambda is the memory address `rbp-0x10`, as opposed to `rbp-0x20` for the functor; they are in this order due to the order in which we declare them.

Summary:

Capture-by-value lambdas work almost identically to a standard functor: they both allocate an object where the captured value is stored and take a hidden function parameter pointing to this object. The code executed by the function call for both the lambda and the functor are the same. The sole difference is that the lambda's constructor is inlined into the function where the lambda is created, rather than being a separate function like the functor's constructor.

Capture By Reference

When capturing by reference, the value captured can be modified in the same manner as a parameter passed by reference to a normal C++ function. As with capturing by value, we can implement this behavior using

either a functor or a lambda:

```
1  class Functor {
2    public:
3      Functor(int& x): m_x(x) {}
4
5      int operator()(int a) {
6        return a + m_x++;
7      }
8
9    private:
10     int& m_x;
11 };
12
13 int main() {
14   int x = 3;
15
16   auto lambda = [&] (int a) { return a + x++; };
17
18   Functor functor(x);
19
20   volatile int y1 = functor(5);
21   volatile int y2 = lambda(5);
22
23   return 0;
24 }
```

Functor:

Again, our functor will have both a constructor and a call operator. We should expect this to behave similarly to the capture-by-value version, and GCC does not disappoint:

```
161 0000000000400540 <Functor::Functor(int&)>:
162     Functor(int& x): m_x(x) {}
163   400540: 55                    push   rbp
164   400541: 48 89 e5              mov    rbp,rsp
165   400544: 48 89 7d f8           mov    QWORD PTR [rbp-0x8],rdi
166   400548: 48 89 75 f0           mov    QWORD PTR [rbp-0x10],rsi
167   40054c: 48 8b 45 f8           mov    rax,QWORD PTR [rbp-0x8]
168   400550: 48 8b 55 f0           mov    rdx,QWORD PTR [rbp-0x10]
169   400554: 48 89 10              mov    QWORD PTR [rax],rdx
170   400557: 90                    nop
171   400558: 5d                    pop    rbp
172   400559: c3                    ret
```

The sole difference from the capture-by-value constructor is that our second parameter is now an 8-byte value, and thus in `rsi` instead of `esi`. You might suspect that this second parameter is a pointer to `x` in `main()`, and you would not be wrong:

```
128   int x = 3;
129   4004ee: c7 45 fc 03 00 00 00  mov    DWORD PTR [rbp-0x4],0x3
...
134   Functor functor(x);
135   4004fd: 48 8d 55 fc           lea    rdx,[rbp-0x4]
136   400501: 48 8d 45 e0           lea    rax,[rbp-0x20]
137   400505: 48 89 d6              mov    rsi,rdx
138   400508: 48 89 c7              mov    rdi,rax
139   40050b: e8 30 00 00 00        call   400540 <Functor::Functor(int&)>
```

Lines 135 and 137 put a pointer to `x` (allocated on line 129) into `rsi`. The remainder of the code sets up the `this` pointer for the constructor, as before.

```
174 000000000040055a <Functor::operator()(int)>:
175     int operator()(int a) {
176   40055a: 55                    push   rbp
177   40055b: 48 89 e5              mov    rbp,rsp
178   40055e: 48 89 7d f8           mov    QWORD PTR [rbp-0x8],rdi
179   400562: 89 75 f4              mov    DWORD PTR [rbp-0xc],esi
180       return a + m_x++;
181   400565: 48 8b 45 f8           mov    rax,QWORD PTR [rbp-0x8]
182   400569: 48 8b 00              mov    rax,QWORD PTR [rax]
183   40056c: 8b 10                 mov    edx,DWORD PTR [rax]
184   40056e: 8d 4a 01              lea    ecx,[rdx+0x1]
185   400571: 89 08                 mov    DWORD PTR [rax],ecx
186   400573: 8b 45 f4              mov    eax,DWORD PTR [rbp-0xc]
187   400576: 01 d0                 add    eax,edx
188     }
189   400578: 5d                    pop    rbp
190   400579: c3                    ret
```

The function call is noticably different from the capture-by-value version, as we'd expect: we are performing both an addition and an increment, and the increment has to propagate back to `main()`. But wait, there's only one `add` instruction! What's going on here?

First, `this` ends up in `rax` via lines 178 and 181. Now, recall that the value there is a pointer to the functor object, which consists of one pointer, `m_x`, to the memory location (of `x`) we passed the constructor. So we're two layers of indirection away from the value we need. Lines 182 and 183 perform the double-dereference needed to copy the value of `x` to `edx`.

Line 184 is an unusual way of writing an increment! Recall that `edx` and `rdx` are the same register; `lea` expects and 8-byte source (since it was meant to do pointer arithmetic and we're on a 64-bit system). The result of adding 1 to `edx` is stored in `ecx`; this is the value `m_x++`. Line 185 copies this value back to where `rax` is pointing—which is `x` in `main()`.

Lastly, lines 179 and 186 copy the second parameter, `a`, into `eax`, and line 187 performs the addition, as in previous versions.

Lambda:

As with capture-by-value, the only difference between the functor and the lambda is that the lambda's "constructor" is inlined into `main()`, on lines 131 and 132:

```
123 00000000004004e6 <main>:
124 int main() {
125   4004e6: 55                    push   rbp
126   4004e7: 48 89 e5              mov    rbp,rsp
127   4004ea: 48 83 ec 30           sub    rsp,0x30
128   int x = 3;
129   4004ee: c7 45 fc 03 00 00 00  mov    DWORD PTR [rbp-0x4],0x3
130   auto lambda = [&] (int a) { return a + x++; };
131   4004f5: 48 8d 45 fc           lea    rax,[rbp-0x4]
132   4004f9: 48 89 45 f0           mov    QWORD PTR [rbp-0x10],rax
```

The code executed when the lambda is called is identical:

```
102 00000000004004c6 <main::{lambda(int)#1}::operator()(int) const>:
108   auto lambda = [&] (int a) { return a + x++; };
109   4004c6: 55                    push   rbp
110   4004c7: 48 89 e5              mov    rbp,rsp
111   4004ca: 48 89 7d f8           mov    QWORD PTR [rbp-0x8],rdi
112   4004ce: 89 75 f4              mov    DWORD PTR [rbp-0xc],esi
113   4004d1: 48 8b 45 f8           mov    rax,QWORD PTR [rbp-0x8]
```

```
4004d5: 48 8b 00             mov    rax,QWORD PTR [rax]
4004d8: 8b 10                mov    edx,DWORD PTR [rax]
4004da: 8d 4a 01             lea    ecx,[rdx+0x1]
4004dd: 89 08                mov    DWORD PTR [rax],ecx
4004df: 8b 45 f4             mov    eax,DWORD PTR [rbp-0xc]
4004e2: 01 d0                add    eax,edx
4004e4: 5d                   pop    rbp
4004e5: c3                   ret
```

Summary:

When capturing by reference, the functor and lambda objects contain a pointer instead of a value, demonstrating that the behavior of references is implemented using pointers under the hood. As with capture-by-value, the functor and lambda call code is equivalent, but the lambda's constructor is inlined, whereas the functor's is not.

Conclusion

C++ lambdas and functors are more similar than they are different. This is to be expected; the main goal of lambdas is to be a syntactically simple means of creating functions and closures. They differ slightly from plain functions, even when no variables are being captured. To summarize the key differences:

1. Functors and lambdas are always passed a `this` pointer, whereas plain functions naturally are not. This consumes an extra register and 8 bytes of stack space.
2. Lambda "constructors" are inlined into the function in which the lambda is created. This significantly reduces the amount of copying performed (2 instructions for lambdas, 5 for functors), as well as avoiding a function call setup and teardown.

Overall, the costs for #1 are minor, and are probably eliminated by an optimization pass in the compiler. The costs for #2 are somewhat higher, but they are cheaper for lambdas! Again, I suspect that an optimization pass would eliminate the difference between the two.

Homework

If you want to learn more, here are some things to investigate:

1. What impact do various optimization levels (`-O1`, `-O2`, `-Os`, and/or `-Og`) have on this analysis?
2. How does the output of Clang differ from that of GCC?