

Noah French (njf5cu)

Lab 9

11/16/17

Filename: postlab.pdf

File Description: Lab 9 Post-lab Report

Dynamic Dispatch

To explore how dynamic dispatch is implemented at the assembly level, I first created a simple class hierarchy in C++.

```
class Mammal {
public:
    virtual string getName() {
        return name;
    }
    virtual int getAge() {
        return age;
    }
private:
    string name;
    int age;
};

class Platypus : Mammal {
public:
    virtual string getName() {
        return "Platypi do not deserve names.";
    }
    virtual int getAge() {
        return 0;
    }
};
```

The platypus is a vile and disgusting creature. It is a mammal, yet it lays eggs? It sweats milk, so a female's offspring must incestuously lick the skin of its mother to feed? Because of this, it's important that at runtime, if the object in question is a platypus, the platypus methods are called. Platypi neither deserve names nor ages, so if the Mammal superclass methods were accidentally called, an error would be thrown.

To compile the .cpp I used the following flags: -m64 -mllvm --x86-asm-syntax=intel -S -fomit-frame-pointer. At first, nothing interesting was generated. The x86 code was under 100 lines, and there was no sign of a virtual method table. Then, I decided to add a main method, wherein I create a Platypus object and call the getName() and getAge() methods on said object:

```
int main() {
    Platypus perry;
    perry.getName();
    perry.getAge();

    return 0;
}
```

```
}
```

With that addition, the x86 code increased in length to around 450 lines. In the C++ code, there's no information stored in either class that explicitly states which class it is. However, it's obvious that at runtime, dynamic dispatch allows the correct methods to be called. This is because a hidden pointer is placed inside each instance of the class that points to the virtual method table (<http://www.drbio.cornell.edu/pl47/programming/TICPP-2nd-ed-Vol-one-html/Chapter15.html>). Here is the assembly code for the instantiation of the Platypus and the invocation of getName()

```
    sub    rsp, 120
.Ltmp11:
    .cfi_def_cfa_offset 128
    mov     dword ptr [rsp + 116], 0
    lea     rax, [rsp + 64]
    mov     rdi, rax
    mov     qword ptr [rsp + 8], rax # 8-byte Spill
    call    _ZN8PlatypusC2Ev
.Ltmp1:
    lea     rdi, [rsp + 32]
    mov     rsi, qword ptr [rsp + 8] # 8-byte Reload
    call    _ZN8Platypus7getNameB5cxx11Ev
```

First, a block of contiguous memory is allocated on the stack to accommodate the Platypus object. Then, 0 is added near the block's beginning. The constructor is called. The first field stored upon calling the constructor is a pointer to the virtual method table. To call the getName method via the virtual method table, the pointer to the virtual method table is loaded into rdi. The getName method is then called via the virtual method table pointer. Inside the virtual method table, the method pointer is looked up, and the method is jumped to in memory.

Interestingly, though it's related to dynamic dispatch, virtual tables themselves are stored in a contiguous block of static memory (<https://yosefk.com/c++fqa/inheritance-virtual.html>).

Optimized Code

To compare assembly code generated without and with the -O2 flag, I first created a simple C++ function that takes in an integer, and uses it to loop through zero and that integer. Every iteration of the loop, the function adds one to an integer that originally starts at 0. It returns the result. The function is, in actual practice, completely useless, because it effectively returns the same integer value that it takes in as a parameter. It should be useful, however, in analyzing how loops and function calls are represented on the assembly level. The function looks like this:

```
int loopFunc(int x) {
    int y = 0;
    for (int i=0; i < x; i++)
        y++;
    return y;
}
```

In the main method of the same .cpp file, I initialize an integer z to equal loopFunc(4).

```
int main() {
    int z = loopFunc(4);
    return 0;
}
```

First, I compiled the .cpp file using the following flags: -m64 -mllvm --x86-asm-syntax=intel -S -fomit-frame-pointer. This is how the loopFunc() function was represented in x86:

```
    mov     dword ptr [rsp - 4], edi
    mov     dword ptr [rsp - 8], 0
    mov     dword ptr [rsp - 12], 0
.LBB1_1:
    mov     eax, dword ptr [rsp - 12]
    cmp     eax, dword ptr [rsp - 4]
    jge     .LBB1_4
    mov     eax, dword ptr [rsp - 8]
    add     eax, 1
    mov     dword ptr [rsp - 8], eax
    mov     eax, dword ptr [rsp - 12]
    add     eax, 1
    mov     dword ptr [rsp - 12], eax
    jmp     .LBB1_1
.LBB1_4:
    mov     eax, dword ptr [rsp - 8]
    retn
```

The loop is clearly evident in the x86 code. Every iteration, the routine jumps back up to the .LBB1_1 marker. One is added to the eax register each iteration. And at the end, the value that had been growing every iteration is moved to the return register, where the value is returned. Here's how the same function was represented in x86 when the -O2 flag was also included:

```
xor     eax, eax
test     edi, edi
cmovns   eax, edi
ret
```

The optimizer flag cleverly recognized what my loop function was doing. Because the loop function ended up returning the input value every time, the assembly code skipped the loop entirely. Instead, it simply took the value from the parameter register, edi, and put it into the return register, edx.

The difference between the x86 code for the caller function (in this case, the main method), was similarly striking. Without the -O2 flag the main method looked like this:

```
    push    rax
.Ltmp1:
    .cfi_def_cfa_offset 16
    mov     edi, 4
    mov     dword ptr [rsp + 4], 0
    call    _Z8loopFunci
```

```

xor    edi, edi
mov    dword ptr [rsp], eax
mov    eax, edi
pop    rcx
ret

```

The value 4 is moved into the parameter register, so it can be passed into the loopFunc(). The resulting value from the function call is stored in memory (int z). The edi register is promptly zeroed, and then that zero is moved to the return register, because the main method always returns 0. The x86 code is distinctly less interesting to analyze when the -O2 flag is used:

```

xor    eax, eax
ret

```

It's clearly evident that the -O2 flag did not deem my task of making `int z = loopFunc(4)` worthy, and (as per the in-lab instruction's warning) it was completely deleted. Presumably, initialize a value is not on its own necessary. These examples exhibit a few key ways that the -O2 flag optimizes code: for one, it looks at the input and output of a function. For one, if the body of the function can achieve the same input → output mapping in fewer steps, it will be rewritten in fewer steps. Second, any unused variables will be completely deleted.

For more compelling examples, I looked at the getNextPrime() method from lab 6 in x86. Without the -O2 flag, the x86 code looked like this:

```

push    rax

.Ltmp39:
    .cfi_def_cfa_offset 16
    mov    dword ptr [rsp + 4], edi
.LBB11_1:
    # =>This Inner Loop Header: Depth=1
    mov    eax, dword ptr [rsp + 4]
    add    eax, 1
    mov    dword ptr [rsp + 4], eax
    mov    edi, eax
    call    _Z10checkprimej
    xor    al, -1
    test   al, 1
    jne    .LBB11_2
    jmp    .LBB11_3
.LBB11_2:
    # in Loop: Header=BB11_1 Depth=1
    jmp    .LBB11_1
.LBB11_3:
    mov    eax, dword ptr [rsp + 4]
    pop    rcx
    ret

```

With the -O2 flag, the code looked like this:

```

    mov    ecx, edi
    .align 16, 0x90
.LBB5_1:

```

```

    inc    ecx
    cmp    ecx, 2
    jb     .LBB5_1
    mov    eax, 2
    je     .LBB5_6
    test   cl, 1
    je     .LBB5_1
    mov    esi, 5
    cmp    ecx, 9
    jb     .LBB5_5
    .align 16, 0x90
.LBB5_8:
    lea    edi, [rsi - 2]
    xor    edx, edx
    mov    eax, ecx
    div    edi
    test   edx, edx
    je     .LBB5_1
    mov    eax, esi
    imul   eax, eax
    add    esi, 2
    cmp    eax, ecx
    jbe    .LBB5_8
.LBB5_5:
    mov    eax, ecx
.LBB5_6:
    ret    # %.loopexit

```

Interestingly, this snippet of code actually got longer with the optimization! However, because massive number of times this function loops to generate prime numbers, even though the optimized code is more lines, it is still more efficient. At runtime, the optimized code will loop far fewer times than the code generated without the -O2 flag. Thus, the optimizer takes a lot of time (and some additional memory) preparing the code at compile-time in order to save a lot of time at run-time.