Justin Choi (jc8mc)
CS 2150 – 106
April 13th 2017

postlab9.pdf

1. Dynamic Dispatch:

Dispatch is the act of invoking a subroutine. Dynamic dispatch is the decision on which member function to invoke made using run-time type of an object while static dispatch is the decision on which member function to using compile-time type of an object. Dynamic dispatch occurs in C++ in the context of inheritance and overloading methods. C++, by default, does not do dynamic dispatch, as it is very slow due to runtime overhead since the program must maintain extra information and the compiler must generate code to determine which member function to invoke while static dispatch is fast. Thus, in order to dynamically dispatch, the virtual keyword must be used. The virtual keyword checks to see if there is a subclass that overwrites a particular method, and if it is overwritten, it uses the overwritten method. Without the virtual keyword, it does static dispatch: it only looks at the type of the pointer, not the pointee. (http://condor.depaul.edu/ichu/csc447/notes/wk10/Dynamic2.htm)

The C++ code below shows static dispatch as the one() method is overwritten but does not use the virtual keyword. Thus, as static dispatch only looks at the type of the pointer, the one() method of the Justin class is called at compile time. It prints out “Justin’s one”

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

class Justin {
public:
  void one();
  void two();
};

class Choi: public Justin {
public:
  void one();
  void two();
};

void Justin::one() {
  cout << "Justin's one" << endl;
}

void Choi::one() {
  cout << "Choi's one" << endl;
}

void Justin::two() {
  cout << "Justin's two" << endl;
}

void Choi::two() {
  cout << "Choi's two" << endl;
}

int main() {
  Justin *j = new Choi();
  j -> one();
  return 0;
}
```

Justin Choi (jc8mc)
CS 2150 – 106
April 13th 2017

The C++ code below shows dynamic dispatch as the one() method is overwritten and uses the virtual keyword. As the virtual keyword checks to see if there is a subclass that overwrites a method, in which the one() method in the Choi subclass overwrites the one() method in the superclass (the Justin class), the one() method in the Choi subclass is called. In the example below, “Choi’s one” is printed.

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

class Justin {
public:
  virtual void one();
  virtual void two();
};

class Choi: public Justin {
public:
  void one();
  void two();
};

void Justin::one() {
  cout << "Justin's one" << endl;
}

void Choi::one() {
  cout << "Choi's one" << endl;
}

void Justin::two() {
  cout << "Justin's two" << endl;
}

void Choi::two() {
  cout << "Choi's two" << endl;
}

int main() {
  Justin *j = new Choi();
  j -> one();
  return 0;
}
```

In this example, the assembly code to the right of us shows the main method with the virtual keyword that uses dynamic dispatch. Rather than calling the superclass Justin, it calls the subclass, Choi, so we know that it uses dynamic dispatch. In the virtual method table, it stores the list of memory addresses and in this case, the overwritten method is called due to virtual keyword.

```
main:                                   # @main
        .cfi_startproc
# BB#0:
        sub     rsp, 24
.Ltmp5:
        .cfi_def_cfa_offset 32
        mov     eax, 8
        mov     edi, eax
        mov     dword ptr [rsp + 20], 0
        call    _Znwm
        xor     esi, esi
        mov     ecx, 8
        mov     edx, ecx
        mov     rdi, rax
        mov     qword ptr [rsp], rax    # 8-byte Spill
        call    memset
        mov     rdi, qword ptr [rsp]    # 8-byte Reload
        call    _ZN4ChoiC2Ev
        mov     rax, qword ptr [rsp]    # 8-byte Reload
        mov     qword ptr [rsp + 8], rax
        mov     rax, qword ptr [rsp + 8]
        mov     rdx, qword ptr [rax]
        mov     rdi, rax
        call    qword ptr [rdx]
        xor     eax, eax
        add     rsp, 24
        ret
```

Thus, we can assume that without the virtual keyword, the superclass's method, Justin's one would be called due to static dispatch.

Dynamic memory is on the heap while static memory is on the stack. The new keyword must be used to allocate memory dynamically and the delete keyword must be used to deallocate memory. Although not shown, malloc can be also used in C++ for the new keyword. When searched online, I saw that dynamic memory in assembly might sometimes have calling malloc to identify the dynamic memory (http://www.linuxquestions.org/questions/linux-software-2/dynamic-memory-allocation-in-assembly-764638/) However, as we are not particularly interested in how dynamic memory works in assembly compared to how dynamic dispatch works in assembly, we can note that in the assembly code above, the dynamically allocated pointer j that has type Justin has no effect whatsoever on overwritten methods such as one() in the example.

Justin Choi (jc8mc)
CS 2150 – 106
April 13th 2017

2. Optimized Code:

For Lab 9: Assembly, Part 2, I chose to answer the 2nd optional question for the in-lab, which said to compare code generated normally compared to optimized code. I first generated assembly code generated normally, without the –O2 compiler flag. Then, to generate the optimized version of the assembly code, I compiled with the –O2 compiler flag.

To explore some features of the optimized –O2 compiler flag, I first tried a code with a simple C++ code that contained loops and function calls in main. I created a C++ code that multiplies the first parameter by the second parameter in my function multiply and called multiply in my main method. I used the intel version of the assembly code.

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

int multiply(int x, int y) {
  int total;
  for (int i = 0; i < y; i++) {
    total += x;
  }
  return total;
}


int main() {
  int a = 25;
  int b = 5;
  int z = multiply(a, b);
  cout << z << endl;
  return 0;
}
```

The non-optimized version is on the left and optimized version is on the right.

```
_Z8multiplyii:                          # @_Z8multiplyii
        .cfi_startproc
# BB#0:
        mov     dword ptr [rsp - 4], edi
        mov     dword ptr [rsp - 8], esi
        mov     dword ptr [rsp - 12], 0
        mov     dword ptr [rsp - 16], 0
.LBB1_1:                                # =>This Inner Loop Header: Depth=1
        mov     eax, dword ptr [rsp - 16]
        cmp     eax, dword ptr [rsp - 8]
        jge     .LBB1_4
# BB#2:                                 #   in Loop: Header=BB1_1 Depth=1
        mov     eax, dword ptr [rsp - 4]
        add     eax, dword ptr [rsp - 12]
        mov     dword ptr [rsp - 12], eax
# BB#3:                                 #   in Loop: Header=BB1_1 Depth=1
        mov     eax, dword ptr [rsp - 16]
        add     eax, 1
        mov     dword ptr [rsp - 16], eax
        jmp     .LBB1_1
.LBB1_4:
        mov     eax, dword ptr [rsp - 12]
        ret
.Lfunc_end1:
        .size   _Z8multiplyii, .Lfunc_end1-_Z8multiplyii
        .cfi_endproc

        .globl  main
        .align  16, 0x90
        .type   main,@function
```

```
main:                                   # @main
        .cfi_startproc
# BB#0:
        sub     rsp, 24
.Ltmp1:
        .cfi_def_cfa_offset 32
        mov     dword ptr [rsp + 20], 0
        mov     dword ptr [rsp + 16], 25
        mov     dword ptr [rsp + 12], 5
        mov     edi, dword ptr [rsp + 16]
        mov     esi, dword ptr [rsp + 12]
        call    _Z8multiplyii
        movabs  rdi, _ZSt4cout
        mov     dword ptr [rsp + 8], eax
        mov     esi, dword ptr [rsp + 8]
        call    _ZNSolsEi
        movabs  rsi, _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
        mov     rdi, rax
        call    _ZNSolsEPFRSoS_E
        xor     ecx, ecx
        mov     qword ptr [rsp], rax    # 8-byte Spill
        mov     eax, ecx
        add     rsp, 24
```

```
_Z8multiplyii:                          # @_Z8multiplyii
        .cfi_startproc
# BB#0:                                 # %._crit_edge
        imul    edi, esi
        xor     eax, eax
        test    esi, esi
        cmovg   eax, edi
        ret
.Lfunc_end0:
        .size   _Z8multiplyii, .Lfunc_end0-_Z8multiplyii
        .cfi_endproc

        .globl  main
        .align  16, 0x90
        .type   main,@function
```

```
main:                                   # @main
        .cfi_startproc
# BB#0:
        push    r14
.Ltmp0:
        .cfi_def_cfa_offset 16
        push    rbx
.Ltmp1:
        .cfi_def_cfa_offset 24
        push    rax
.Ltmp2:
        .cfi_def_cfa_offset 32
.Ltmp3:
        .cfi_offset rbx, -24
.Ltmp4:
        .cfi_offset r14, -16
        mov     edi, _ZSt4cout
        mov     esi, 125
        call    _ZNSolsEi
        mov     r14, rax
        mov     rax, qword ptr [r14]
        mov     rax, qword ptr [rax - 24]
        mov     rbx, qword ptr [r14 + rax + 240]
        test    rbx, rbx
        je      .LBB1_5
```

Justin Choi (jc8mc)
CS 2150 – 106
April 13th 2017

As I generated a non-optimized version and an optimized version, I recognized that for my function multiply, which is the callee as it is called by main, the caller, I recognized the it was generally more compact and concise. I intentionally made the multiply function very inefficient. I could have not used a for loop and just multiplied the two values. When compiled without the optimized -O2 flag, it does a faithful but literal translation into assembly code, which causes longer time when it is executed. The optimized version gets rid of unnecessary code. (http://www.agner.org/optimize/optimizing_assembly.pdf)

First, I recognized that in the callee function, multiply, there is a lot of copying of values going on in the non-optimized version. However, the callee function of the optimized version did only had one copying of value, the cmovg instruction opcode. As Professor Bloomfield mentioned in lecture about his max function, I noticed the cmovg instruction opcode generated in the optimized version, which is for “conditional move if greater than,” which will move the greater value into the first parameter. So, edi is moved into eax as edi is greater than eax. The two parameters (edi and esi) are multiplied while eax is zeroed out. Compared to the non-optimized version, there were numerous copying of values in order to finally move the final value into eax. This reduces the running time in the optimized version, as there are fewer instructions to copy values from one register to another.

Second, I recognized that there were local variables created onto the stack for copying and moving values in the non-optimized version. As there is an int total, a local variable, in my multiply function, which is totally unnecessary, if you only care about speed for running it faster. This creation of local variables and copying of values took more instructions compared to the optimized version. The optimized version just got rid of the unnecessary local variables such as int total since the optimized flag wants to reduce the running speed and wants to make it faster.

Third, as the optimized version of the assembly code does not need a local variable int total that causes a lot of copying of values and moving values in the non-optimized version, I realized that in the optimized version, it simply just multiplies the first parameter and the second parameter with the imul opcode instruction, which multiplies the two parameters. It creates a more concise version and takes fewer opcodes to do the whole operation that is created in the very inefficiently in the non-optimized version: it uses a for loop to add the first parameter to the local variable, int total, which does the same thing as multiplying but has more instructions.

Fourth, I also recognized that there is memory access in the non-optimized version since it needs to copy and move values. And if we assume 3 cycles per memory access, there are about 33 cycles as there are 11 memory accesses in the non-optimized version (3 * 11), which takes more time and is inefficient. For a cycle, the CPU has to stall and wait for it until it is over. However, the optimized version did not have any memory accesses, which reduces the time. The optimized version only takes 4 or 5 cycles since it doesn't have memory accesses.

In conclusion, the optimized version creates more compact and concise code compared to non-optimized version. Other than these differences that you can see in the assembly code, Bloomfield also mentioned in lecture that optimized code for very large programs would take longer time to compile since it wants to optimize and shorten the code in order to improve running time. Also, he mentioned that as there are no local variables created in the optimized code, it is very hard to debug when it is compiled with the –O2 flag since it is hard to trace what is going on in each level. Thus, for speed of execution, optimized code wins but for debugging and compile time, non-optimized code is better.