Derek Johnson, dej3tc, 11/18/19
Optimized Code:

In order to show the changes at the assembly level that will occur with code optimization I created three simple methods in C++. The first is an adder method that takes in an int t and outputs t += 1. The second method, called point, takes in and integer and outputs a pointer to that integer. The third and final method, main, takes no arguments and performs several operations on a counter variable. The operations are shown in the appendix below.

It is immediately obvious that the -O2 flag has significantly altered the assembly code in 4 main ways.

1) The first and most apparent is the reduction in operations. The unoptimized code is 43 lines long while the optimized code is only 9. While it is not always true that less lines mean faster runtimes, in general this will hold. Additionally, I noticed from viewing other code optimizations that this is not a normal outcome. There are many situations in which the optimized code will actually be longer than the unoptimized code in terms of # of commands present. These “longer” programs can still be faster however due to reduction of loops and other intense commands.
2) In the main method, loop unrolling is used to speed up the process that occurs in the for loop. Instead of adding two to the counter 6 times as will happen in the unoptimized assembly, the final value after all of the operations will be loaded into the return register. The compiler can do this because this value will remain constant for all runs of the program.
3) For the adder method the main difference between the optimized and unoptimized code is the command used. In the unoptimized code, the parameter was loaded into a spot on the stack and then the add command was used to add 1 to it. In the optimized code, the lea command was used to perform addition and movement into the return register in one command. We explored using the lea command in the prelab and it seems to be a much more logical form of arithmetic than the built in commands.
4) The point function is a fairly illogical method and because of this the optimized assembly does nothing except xor the return value. The unoptimized assembly will make commands for every action you write, no matter how illogical.

Sources:
https://www.godbolt.org/
https://en.wikipedia.org/wiki/Category:Compiler_optimizations
https://www.geeksforgeeks.org/code-optimization-in-compiler-design/
https://compileroptimizations.com/

APPendix:

C++ Code

```cpp
int adder(int c){
    return c+=1;
}

int * point(int p){
    int * t =  &p;
    return t;
}

int main() {
  const int two = 2;
  int counter = 0;
  for(int i=0; i<6;i++) {
    counter += two;
  }
  counter = adder(counter);
  int * p = point(counter);
  counter = counter^two;
  return counter;
}
```

Unoptimized Compilation

```asm
adder(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        add     DWORD PTR [rbp-4], 1
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
point(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        lea     rax, [rbp-20]
        mov     QWORD PTR [rbp-8], rax
        mov     rax, QWORD PTR [rbp-8]
        pop     rbp
        ret
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     DWORD PTR [rbp-12], 2
        mov     DWORD PTR [rbp-4], 0
        mov     DWORD PTR [rbp-8], 0
.L7:
        cmp     DWORD PTR [rbp-8], 5
        jg      .L6
        add     DWORD PTR [rbp-4], 2
        add     DWORD PTR [rbp-8], 1
        jmp     .L7
.L6:
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    adder(int)
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    point(int)
        mov     QWORD PTR [rbp-24], rax
        xor     DWORD PTR [rbp-4], 2
        mov     eax, DWORD PTR [rbp-4]
        leave
        ret
```

Optimized Compilation

```asm
adder(int):
        lea     eax, [rdi+1]
        ret
point(int):
        xor     eax, eax
        ret
main:
        mov     eax, 15
        ret
```

Dynamic dispatch

Dynamic Dispatch is the process that allows us to implement polymorphic operations that are called at run time instead of compile time. Polymorphism is a fairly central concept to object oriented coding as it allow for operation override in subclasses. In most cases, the compiler will be able to decide what class an object is by its type declaration. There are situations however where we may declare a parent class object and later instantiate it with the constructor of a child class. In this situation, conflict could arise during method calls. In order to handle these conflicts we use virtual functions. With this process, the operation bonding does not happen until run time. This is a powerful feature as it allows us more flexibility in object creation. To implement a virtual function, the compiler will create a virtual function table and a pointer from the object constructed to that table. This way, when an operation is called on an object, that object will follow its pointer to the virtual table and from there determine the specific operation to perform. Because of this, virtual methods incur an increase in memory and run time,although both are very small.

When we compare dynamic dispatch to static dispatch in assembly we can see that dynamic uses a more complicated system for running methods but it still calls in the same way. In the example below, Dog is a subclass of Animal and both have the methods getType and getName. Both of these methods are labeled virtual. If we were to call the getType() method for dog using static dispatch, the assembly would do it like this,

```
call getType
```

Instead, the call looks like this.

```
mov   rax, QWORD PTR [rbp-24]
mov   rax, QWORD PTR [rax]
mov   rcx, QWORD PTR [rax]
call  rcx
```

Instead of calling a method, it is calling a spot in memory. The spot it is calling is the vtable for Dog: It then loads the function address from the vtable and calls the function indirectly.

Sources:
https://en.wikipedia.org/wiki/Dynamic_dispatch
https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/
https://condor.depaul.edu/ichu/csc447/notes/wk10/Dynamic2.htm
https://www.godbolt.org/

Code:

C++ Code

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

class Animal {
  public:
    virtual string getType(){
      return "Animal";
    }
    virtual string getName(){
      return "UnNamed";
    }
};

class Dog : public Animal {
  private:
    string name;
  public:
  Dog(const string& d_name){
    name = d_name;
  }
  void setName(const string& d_name){
    name = d_name;
  }
  virtual string getType(){return "Dog";}
  virtual string getName(){return name;}
};

int main() {
  Animal* a;
  Animal* b;
  a = new Animal();
  b = new Dog("Fido");
  string s1 = a->getType();
  string s2 = b->getType();
  string s3 = a->getName();
  cout << s1 << endl;
  cout << s2 << endl;
  return 0;
}
```

Assembly:
Code for the call a->getType:

```asm
        mov     rax, QWORD PTR [rbp-24]
        mov     rax, QWORD PTR [rax]
        mov     rcx, QWORD PTR [rax]
        lea     rax, [rbp-112]
        mov     rdx, QWORD PTR [rbp-24]
        mov     rsi, rdx
```

```
        lea     rax, [rbp-112]
        mov     rdi, rax
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >
        mov     eax, ebx
        jmp     .L37
        mov     r12, rax
```

```
.L30:
        lea     rax, [rbp-112]
        mov     rdi, rax
        call    std::__cxx11::basic_string<char, std::char_
        mov     rax, rbx
        mov     rdi, rax
        call    _Unwind_Resume
.L37:
```

VTAble info:

```
vtable for Dog:
        .quad   0
        .quad   typeinfo for Dog
        .quad   Dog::getType[abi:cxx11]()
        .quad   Dog::getName[abi:cxx11]()
vtable for Animal:
        .quad   0
        .quad   typeinfo for Animal
        .quad   Animal::getType[abi:cxx11]()
        .quad   Animal::getName[abi:cxx11]()
typeinfo for Dog:
        .quad   vtable for __cxxabiv1::__si_class_type_info+16
        .quad   typeinfo name for Dog
        .quad   typeinfo for Animal
typeinfo name for Dog:
        .string "3Dog"
typeinfo for Animal:
        .quad   vtable for __cxxabiv1::__class_type_info+16
        .quad   typeinfo name for Animal
typeinfo name for Animal:
        .string "6Animal"
__static_initialization_and_destruction_0(int, int):
```