Joseph Elsisi jre3wjh
Lab 108
November 21
Inheritance in C++:

My two class:

```cpp
class Animal {
    public:
        Animal(void){lives = 0;}
        ~Animal(){}
        void setLives(int val){
            lives = val;
        }
        int getLives(){
            return lives;
        }
        int eyes = 2;
    private:
        int lives;
};
```

```cpp
class Cat: public Animal{
    public:
        Cat(void){claws = 0; setLives(9);}
        ~Cat(){}
        void setClaws(int claw){
            claws = claw;
        }
        int getClaws(){
            return claws;
        }
    private:
    int claws;
};
```

My main:

```cpp
int main(){
    Animal crossing;
    crossing.setLives(1);
    Cat furball;
    furball.setLives(7);
    furball.setClaws(5);
    funcDoNothing(crossing, furball);
    return 0;
}
```

As can be seen here, two objects are created, an Animal called crossing and a Cat that inherits from animal called furball. They each have public and private data members, with Cat's *lives* data field being inherited from Animal. What a Cat can do that an animal can't is store the number of *claws* it has. This incredible resource https://www.drdobbs.com/embedded-systems/object-oriented-programming-in-assembly/184408319 helped to explain a lot of what I will investigate.

To understand how both the parent and child classes in this scenario are implemented, I looked at this snippet of code:

```asm
Cat::Cat() [base object constructor]:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        mov     rdi, rax
        call    Animal::Animal() [base object constructor]
        mov     rax, QWORD PTR [rbp-8]
        mov     DWORD PTR [rax+8], 0
        mov     rax, QWORD PTR [rbp-8]
        mov     esi, 9
        mov     rdi, rax
        call    Animal::setLives(int)
        nop
        leave
        ret
```

Cat, the child class first subs 16 from the stack pointer, meaning that it is reserving 16 bytes of space for the object. Some additional calls are made before the constructor of Animal is called. This shows that all object oriented programming really is is another abstraction for the programmer. Instead of having to rewrite everything in the Animal class into a Cat class, the programmer can pull all the data and functionality they need to make the Cat child class. In memory, a child class will have the data of the parent class first, and right after it it'll have its own data. After the Animal information has been created, one can see that the data members specific to Cat, like claw = 0 are then handled. 9 is moved to edi as a parameter to make the Cat's lives, a private field in it's Animal data, to 9.

When the programmer creates an instance of the object, the constructor of that object is called.

```
        lea     rax, [rbp-44]
        mov     rdi, rax
        call    Animal::Animal() [complete object constructor]
        lea     rax, [rbp-52]
        mov     esi, 99
        mov     rdi, rax
        call    Animal::Animal(int)
```

I made a second constructor that takes in an integer. The creation of a new instance differers simply by which constructor is called in assembly.

As for the destructor, when the function funcDoNothing is finished, and the two newly created objects have run out of scope, their destructors are called.

```
void funcDoNothing(Animal an, Cat at){
    Animal john = an;
    Cat sydney = at;
    john.getLives();
    sydney.getLives();
    int takeSpace;
}
int main(){
    Animal crossing;
```

```
105         mov     rdi, rax
106         call    Animal::getLives()
107         lea     rax, [rbp-20]
108         mov     rdi, rax
109         call    Animal::getLives()
110         lea     rax, [rbp-20]
111         mov     rdi, rax
112         call    Cat::~Cat() [complete object destructor]
113         lea     rax, [rbp-8]
114         mov     rdi, rax
115         call    Animal::~Animal() [complete object destructor]
```

How the destructor actually works is a bit difficult to understand and there are minimal resources explaining this concept. What is first done, as seen on line 112 in the above snippit, is that the address of the start of the Cat object is loaded into rdi. It is the parameter of ~Cat(). From there,

```
Cat::~Cat() [base object destructor]:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        mov     rdi, rax
        call    Animal::~Animal() [base object destructor]
        nop
        leave
        ret
```

all the Cat contents are cleared. The same address is then passed into ~Animal() and the same process of clearing that memory is done for the animal data.

Dynamic Dispatch:
Big class is the parent of Small class.

```cpp
class Big{
    public:
        int sandwich = 1;
        bool status(){
            return sandwich;
        }
};
class Small: public Big{
    public:
    int sandwich = 0;
    bool status(){
        return sandwich;
    }
```

```cpp
int main(){
    Big * ptr;
    int input;
    cin >> input;

    if(input > 0){
        ptr = new Big;
    } else {
        ptr = new Small;
    }
    if((ptr -> status())){
        cout << "BIG";
    }
}
```

To investigate dynamic dispatch, I first wrote a program that must determine what method to call at runtime without the help of the virtual keyword. Unsurprisingly, the subroutine of the parent class is called, despite there being ambiguity over the type of the object being called on.

```asm
        mov     rax, QWORD PTR [rbp-24]
        mov     rdi, rax
        call    Big::status()
        test    al, al
        je      .L8
```

Now to see dynamic dispatch in action, I changed how I implemented the methods in each class, adding the virtual keyword to their definitions. The difference is stark.

```asm
        mov     rax, QWORD PTR [rbp-24]
        mov     rax, QWORD PTR [rax]
        mov     rdx, QWORD PTR [rax]
        mov     rax, QWORD PTR [rbp-24]
        mov     rdi, rax
        call    rdx
        movzx   eax, al
        test    eax, eax
        sete    al
        test    al, al
        je      .L10
```

As can be seen in this snippet, many more commands are needed for the program to determine which method should be called. The first thing to notice here is that no call to a specific method is made. Instead, the address stored in rdx is called. This post confirms my understanding.
https://stackoverflow.com/questions/9995922/how-to-tell-if-a-program-uses-dynamic-dispatch-by-looking-at-the-assembly
The address that is called is loaded from the first block of memory in the object being worked with. This block is a pointer to the objects virtual method table.

The method table will have a pointer to the correct implementation of the method for the given object in it, and will call that method. This requires a lot of dereferencing, so it makes sense that I had to force the compiler, through the virtual keyword, to do this operation.