Riya Baphna
Rb4nk
04/11/19
Postlab9

1. **Inheritance**
   I first created an instance of Inheritance in C++ through the Shape and Square class. This included a constructor for each class, a function that sets the name of the shape being created and then a function that prints it out in the shape class, and also a function that calculates the area based on an int set in the Square class, which inherits from Shape. The print function in the Square class also calls the print function in the Shape class.

```
1
2   #include <cstdlib>
3   #include <iostream>
4   #include <string>
5
6   using namespace std;
7   class Shape{
8       private:
9           string myShape;
10
11      public:
12          Shape (void): myShape (" "){}
13          void setShape (string s){
14              myShape = s;
15          }
16          void print(void) {
17          cout << myShape << endl;
18      }
19
20
21  };
22
23
24  class Square : public Shape{
25  private:
26      int area;
27  public:
28      Square (void){
29          area = 0;
30      }
31      void setArea(int x){
32          area = x*4;
33      }
34
35      void print(void) {
36          Shape :: print();
37          cout << area << endl;
38      }
39
40
41  };
42
43
44  int main(void) {
45      Square sq;
46      sq.setShape("Square");
47      sq.setArea(2);
48      sq.print();
49  }
```

Next I converted the C++ code to assembly. Here is an excerpt of the assembly code in the Square class, specifically in the print method where the square print calls the print from Shape. This is shown in three statements, mov rax, QWORD PTR [rbp-8] ; mov rdi, rax and finally call Shape::print(). The mov keyword allocates space on the stack, then calls the print from Shape to save that string onto the stack.

```
push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        mov     rdi, rax
        call    Shape::print()
        mov     rax, QWORD PTR [rbp-8]
        mov     eax, DWORD PTR [rax+32]
        mov     esi, eax
        mov     edi, OFFSET FLAT:_ZSt4cout
        call    std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
        mov     esi, OFFSET FLAT:_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
        mov     rdi, rax
```

Next, the assembly code for the main method in which a square object created is shown below.

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

As seen, this piece of assembly code calls the Shape constructor when a Square object is constructed. Thus the inheritance is shown in assembly just as it is in C++, in which the Shape class is called when a square object is constructed. The order of the assembly calls follow the order in C++.

2. **Dynamic Dispatch**

It is the process of deciding which member function to invoke and call at run time. The method has to be declared as virtual. A virtual function table is used by the compiler in C++ to implement dynamic dispatch. To do the same in assembly, you would load the address of the virtual function table, then load the function address, then call the function.

Here is a C++ implementation of Dynamic Dispatch including a Sport class, followed by a Basketball class. Both share the methods getScore() but include their own methods basket and gameOver. First is the Sport class:

```
class Sport{
    public:
        int score=5;


      virtual void getScore(){
        cout<< "The score is: "<< score <<endl;
      }
      virtual void gameOver(){
        if (score == 20){
            cout<< "Game over"<<endl;
        }
      }


};
```

Followed by the Basketball class (go hoos) :

```
class Basketball : public Sport{
public:
    virtual void getScore(){
        cout  << "The Score is : "<<score*2<<endl;

    };B
    virtual int basket(){
        score+=2;
        return score;

    }

};
```

Then the main method, that creates the object and calls the methods

```
int main(){
    Sport *b = new Basketball();
    b -> getScore();
    b -> gameOver();
}
```

Following is some assembly code that replicates the portion of code in which the object is created in the main method:

```
mov     edi, 16
call    operator new(unsigned long)
mov     rbx, rax
mov     QWORD PTR [rbx], 0
mov     DWORD PTR [rbx+8], 0
mov     rdi, rbx
call    Basketball::Basketball() [complete object constructor]
mov     QWORD PTR [rbp-24], rbx
```

And also part of the code that creates the Basketball class that inherits from the Sport class.

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
mov     QWORD PTR [rbp-8], rdi
mov     rax, QWORD PTR [rbp-8]
mov     rdi, rax
call    Sport::Sport() [base object constructor]
mov     edx, OFFSET FLAT:vtable for Basketball+16
mov     rax, QWORD PTR [rbp-8]
mov     QWORD PTR [rax], rdx
nop
leave
ret
```

Thus we can see that generating Dynamic Dispatch in assembly code required calling the initial class in the new class that inherits - as seen in call Sport::Sport() in Basketball. We can also see that for the code to be generated in assembly, first the address of the table is loaded into the object using mov and the base pointer rbp, then the function address into the object, then the call to the function using the call instruction.