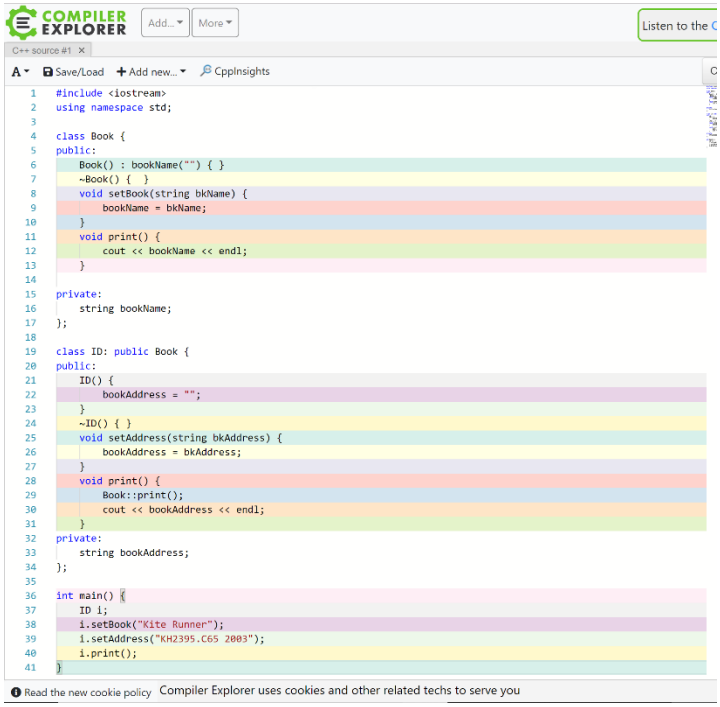
Eza Rasheed
er6qt
04-10-19
postlab9.pdf

Inheritance:

Below, I wrote a C++ code based off of the example given in the lecture slides in order to examine the assembly code for multiple inheritance. ID is the "child class of Book and it inherits the data member bookName field from the Book parent class. In the code below, you can see that both the Book class and ID class have a constructor, destructor (~), a print method, and a private field (bookName, bookAddress). The data members inherited by ID from Book is the bookName field and it includes data member of its own, which is the bookAddress field. In the main method, I created the object "ID i", initialized values into the private fields of bookName and bookAddress and then called the print method on the object the private fields were being called on.

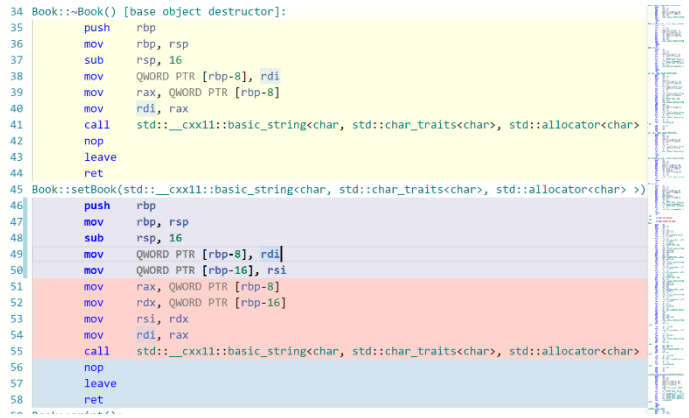


Using the compiler explorer, I generated the assembly code to see where in memory the data members were being laid out and stored in the ID object. Looking at when the private fields in the main method are being stored, you can see that in order to store the bookName and bookAddress fields, space is made on the stack; This space stores the bookName field inherited from the Book class and the bookAddress, which is in the ID class itself. ID assigns space on the stack to call the bookName field (1). Also, the bookName field, which is inherited by the ID class from the Book class is allocated and stored (2), and the bookAddress field in the ID class itself has space allocated on the stack to store the book's address (3).

1.

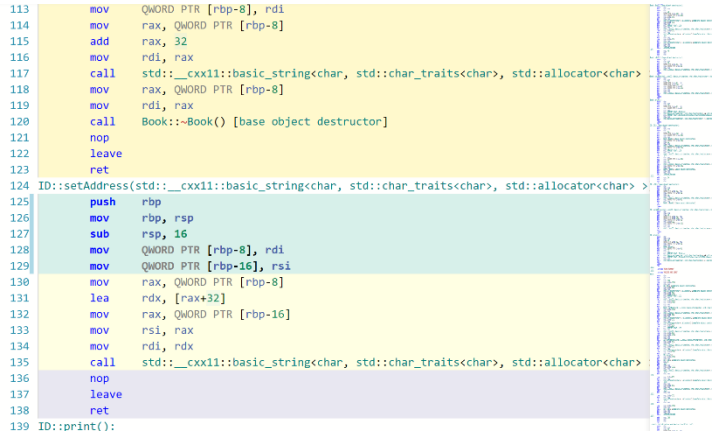
```
ID::ID() [base object constructor]:
        push    rbp
        mov     rbp, rsp
        push    rbx
        sub     rsp, 24
        mov     QWORD PTR [rbp-24], rdi
        mov     rax, QWORD PTR [rbp-24]
        mov     rdi, rax
        call    Book::Book() [base object constructor]
        mov     rax, QWORD PTR [rbp-24]
        add     rax, 32
        mov     rdi, rax
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
        mov     rax, QWORD PTR [rbp-24]
        add     rax, 32
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, rax
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
        jmp     .L11
        mov     rbx, rax
        mov     rax, QWORD PTR [rbp-24]
        add     rax, 32
        mov     rdi, rax
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
        mov     rax, QWORD PTR [rbp-24]
        mov     rdi, rax
        call    Book::~Book() [base object destructor]
        mov     rax, rbx
        mov     rdi, rax
        call    _Unwind_Resume
```

2.



```
Book::~Book() [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    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
        nop
        leave
        ret
Book::setBook(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     QWORD PTR [rbp-8], rdi
        mov     QWORD PTR [rbp-16], rsi
        mov     rax, QWORD PTR [rbp-8]
        mov     rdx, QWORD PTR [rbp-16]
        mov     rsi, rdx
        mov     rdi, rax
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
        nop
        leave
        ret
```

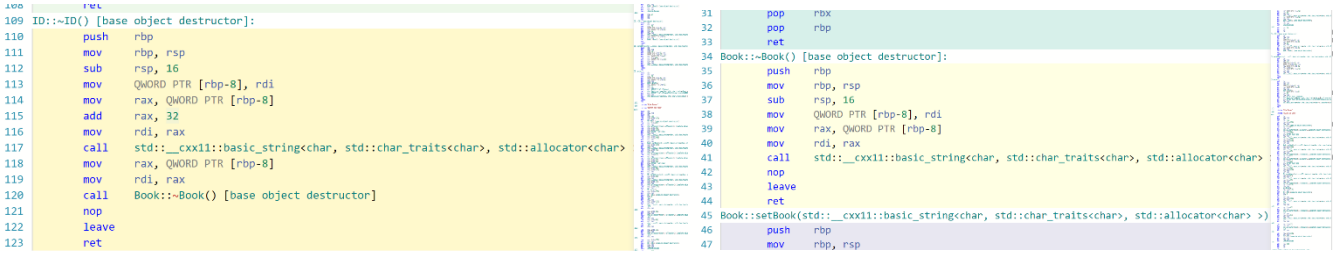
3.



```
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        add     rax, 32
        mov     rdi, rax
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
        mov     rax, QWORD PTR [rbp-8]
        mov     rdi, rax
        call    Book::~Book() [base object destructor]
        nop
        leave
        ret
ID::setAddress(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     QWORD PTR [rbp-8], rdi
        mov     QWORD PTR [rbp-16], rsi
        mov     rax, QWORD PTR [rbp-8]
        lea     rdx, [rax+32]
        mov     rax, QWORD PTR [rbp-16]
        mov     rsi, rax
        mov     rdi, rdx
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
        nop
        leave
        ret
ID::print():
```

The construction and destruction of objects happens in this class hierarchy by first, calling the Book class's constructor to initialize the data members which will be inherited from this parent Book class. Then the ID class's constructor is called to initialize the new data members found within the instance of the object (ID i) itself, in this case bookAddress. When a user-defined object is instantiated, the default constructor is called to allocate space in memory for the two private fields called on the object. When it goes out of scope, the destructor is called for the class object to deallocate memory and "do other cleanup for a class object and its class members when the object is out of scope or explicitly deleted(destroyed)." This will reallocate space taken up

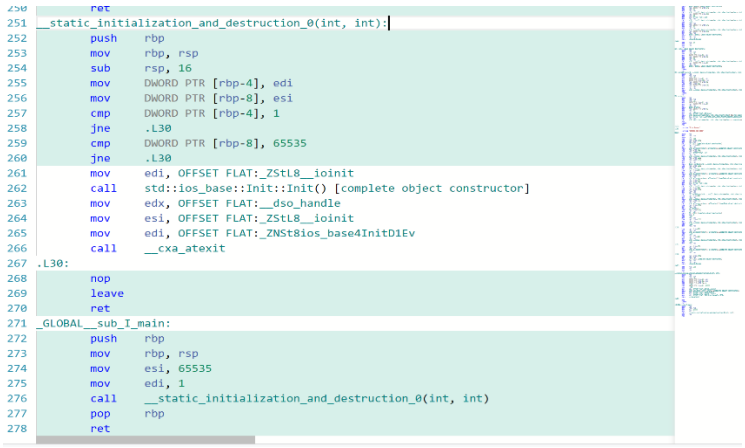
by every object, which then allows space to be freed up in memory and able to be used up again. This process in assembly code using a simple class hierarchy is nearly the same as written in the C++ code. First the object is instantiated, and then the constructor in the parent class (Book) is called, followed by the child class (ID). When the whole function is done running, the destructor of each class is called on the two private fields, bookName and bookAddress. In the screenshot at the end, you can see the assembly code of the class hierarchy for this.



```
ID::~ID() [base object destructor]:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        add     rax, 32
        mov     rdi, rax
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
        mov     rax, QWORD PTR [rbp-8]
        mov     rdi, rax
        call    Book::~Book() [base object destructor]
        nop
        leave
        ret
```

```
        pop     rbx
        pop     rbp
        ret
Book::~Book() [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    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
        nop
        leave
        ret
Book::setBook(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
        push    rbp
        mov     rbp, rsp
```

The screenshots above show the assembly code for when the destructors are being implemented. At the end of the main, static initialization and destruction is called, which is when the destructors and constructors for both the classes are called. Destructors are called in a reverse order from how the constructors get called, for a reason I am not too sure of; first, the ID class's destructor is called, and then, the Book class's destructor is called. Looking at the assembly code below, you can see that the destructors and constructors are getting called at the end of the main function.

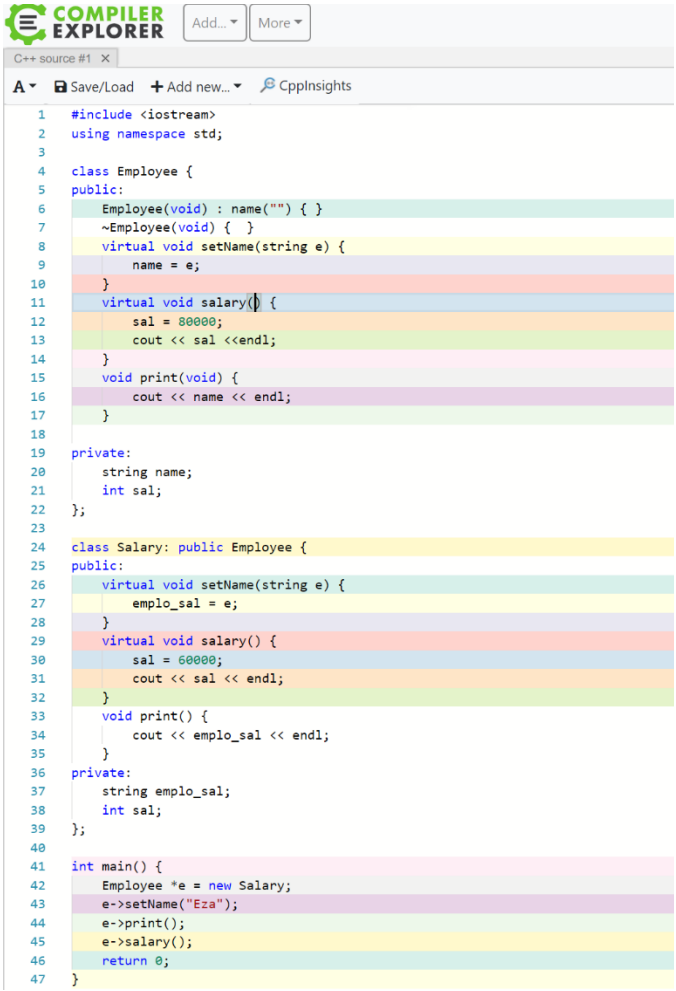


```
        ret
__static_initialization_and_destruction_0(int, int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        cmp     DWORD PTR [rbp-4], 1
        jne     .L30
        cmp     DWORD PTR [rbp-8], 65535
        jne     .L30
        mov     edi, OFFSET FLAT:_ZStL8__ioinit
        call    std::ios_base::Init::Init() [complete object constructor]
        mov     edx, OFFSET FLAT:__dso_handle
        mov     esi, OFFSET FLAT:_ZStL8__ioinit
        mov     edi, OFFSET FLAT:_ZNSt8ios_base4InitD1Ev
        call    __cxa_atexit
.L30:
        nop
        leave
        ret
_GLOBAL__sub_I_main:
        push    rbp
        mov     rbp, rsp
        mov     esi, 65535
        mov     edi, 1
        call    __static_initialization_and_destruction_0(int, int)
        pop     rbp
        ret
```

Dynamic Dispatch:

Dynamic dispatch is implemented when deciding which member function of the subclass (inherited by the super superclass) to invoke using the run-time of an object. It is activated using the "virtual" keyword. If "virtual" is mentioned, the "compiler knows to check the subclass for another version of that function (call method based on what pointer is pointing to)". When using dynamic dispatch, each object contains a pointer to the virtual method table (VMT). This table holds the addresses of the methods. When calling the virtual function, first, the virtual method follows the pointer to that object, second, the object has a pointer that points to a virtual method table, and third, it finds the method in the table and jump to that method. Because object have yo keep track of the VMT, dynamic dispatch incurs runtime overhead, as the program has to

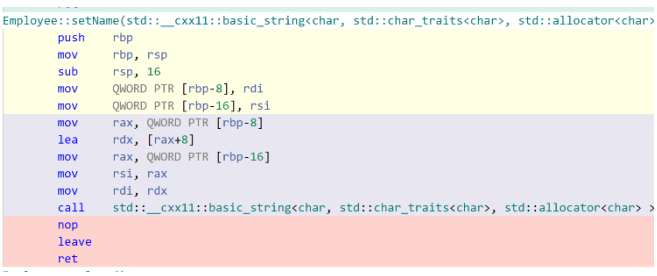
maintain extra information and the “compiler has to generate code to determine which member function to invoke.” Additionally, it requires checking time to see if the subclass has redefined any functions/attributes from the superclass during runtime. Therefore, it is slower than static dispatch. Static dispatch is the default in C++. Unlike dynamic dispatch, it decides on which member function to invoke using the compile-time type of an object. It is faster because the compiler knows what method it will be calling at runtime. To further investigate on dynamic dispatch, I created two classes, similar to the ones I made for inheritance with a bit of modification. I have a superclass called Employee and subclass that inherits from it called Salary. Both classes have a function called setName(), salary(), and print(). Dynamic dispatch is being implemented in the parts of the code where the keyword “virtual” is being used, so when I am creating my functions (lines: 8, 11, 26, 29). It is because of this keyword that dynamic dispatch will know to which member classes to invoke during run time.



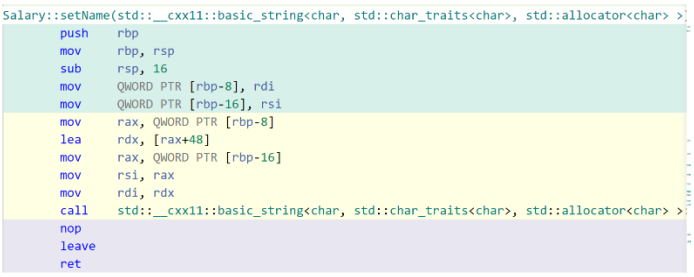
In the screenshot above, in the main method, I created a pointer “e” of type Employee. The pointee will check to see if there is any function of the same name in the subclass as in the superclass. If there is, which in my case there is, the program will run the overridden function in the subclass. For example, when e points to new Salary, it will be checked to see if there is

another function with the same name (setName(), salary()) in the subclass Salary before choosing whether to run that same function from Employee or Salary. Hence why dynamic dispatch decides which data members to invoke using the run time of an object.

In the third screenshot below, you can see how dynamic dispatch calls on data members for implementation in the main method, during run time. Wherever my two classes defined their virtual functions, the “compilers added a hidden member variable to the class that points to an array of pointers to virtual functions” of the virtual method table. These pointers are used during run time to invoke the right function because at compile time, “it may not be known if the base function is to be called or a derived one implemented by a class that inherits from the base class.” Although the assembly code is nearly the same at the machine level for both classes for the setName and salary functions (look at side by side comparison below), the assembly code in the third screenshot below for the main method at runtime shows that the Salary class is called when the setName function is called. This explains how the “virtual” keyword in dynamic dispatch allows the compiler to know to use the subclass’s derived function, implemented by inheriting from the base class, instead of the one in the base class, Employee. Therefore, in the VMT, the value stored in that portion of memory will reference the memory address of the subclass’s (Salary) function (setName()), rather than the base class’s (Employee) function.

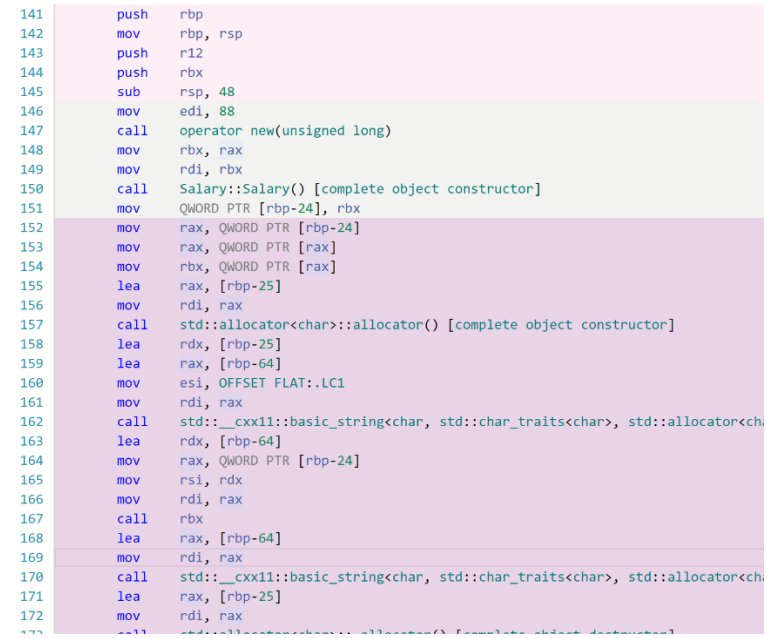


```
Employee::setName(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     QWORD PTR [rbp-8], rdi
        mov     QWORD PTR [rbp-16], rsi
        mov     rax, QWORD PTR [rbp-8]
        lea     rdx, [rax+8]
        mov     rax, QWORD PTR [rbp-16]
        mov     rsi, rax
        mov     rdi, rdx
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >
        nop
        leave
        ret
```



```
Salary::setName(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     QWORD PTR [rbp-8], rdi
        mov     QWORD PTR [rbp-16], rsi
        mov     rax, QWORD PTR [rbp-8]
        lea     rdx, [rax+48]
        mov     rax, QWORD PTR [rbp-16]
        mov     rsi, rax
        mov     rdi, rdx
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >
        nop
        leave
        ret
```

** Main method: Salary class called on line 150, setName() function implemented starting line 152 **



```
140 main:
141         push    rbp
142         mov     rbp, rsp
143         push    r12
144         push    rbx
145         sub     rsp, 48
146         mov     edi, 88
147         call    operator new(unsigned long)
148         mov     rbx, rax
149         mov     rdi, rbx
150         call    Salary::Salary() [complete object constructor]
151         mov     QWORD PTR [rbp-24], rbx
152         mov     rax, QWORD PTR [rbp-24]
153         mov     rax, QWORD PTR [rax]
154         mov     rbx, QWORD PTR [rax]
155         lea     rax, [rbp-25]
156         mov     rdi, rax
157         call    std::allocator<char>::allocator() [complete object constructor]
158         lea     rdx, [rbp-25]
159         lea     rax, [rbp-64]
160         mov     esi, OFFSET FLAT:.LC1
161         mov     rdi, rax
162         call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<ch
163         lea     rdx, [rbp-64]
164         mov     rax, QWORD PTR [rbp-24]
165         mov     rsi, rdx
166         mov     rdi, rax
167         call    rbx
168         lea     rax, [rbp-64]
169         mov     rdi, rax
170         call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<ch
171         lea     rax, [rbp-25]
172         mov     rdi, rax
```

To see the difference in assembly code between dynamic dispatch and static dispatch, I deleted the keyword “virtual” in my functions. The modified C++ code below showed 190 lines of assembly code, as opposed to the 260 lines produced by dynamic dispatch. As mentioned before, this is because in static dispatch, the program decides on which member function to invoke using the compile-time type of an object. Therefore, there are no pointers to/in the VMT needed to be checked and ran through, which saves time.

```
A  Save/Load  + Add new...  CppInsights
1  #include <iostream>
2  using namespace std;
3
4  class Employee {
5  public:
6      Employee(void) : name("") { }
7      ~Employee(void) {  }
8      void setName(string e) {
9          name = e;
10     }
11     void salary() {
12         sal = 80000;
13         cout << sal <<endl;
14     }
15     void print(void) {
16         cout << name << endl;
17     }
18
19 private:
20     string name;
21     int sal;
22 };
23
24 class Salary: public Employee {
25 public:
26     void setName(string e) {
27         emplo_sal = e;
28     }
29     void salary() {
30         sal = 60000;
31         cout << sal << endl;
32     }
33     void print() {
34         cout << emplo_sal << endl;
35     }
36 private:
37     string emplo_sal;
38     int sal;
39 };
40
41 int main() {
42     Employee *e = new Salary;
43     e->setName("Eza");
44     e->print();
45     e->salary();
46     return 0;
47 }
```

```
182         ret
183 _GLOBAL__sub_I_main:
184         push    rbp
185         mov     rbp, rsp
186         mov     esi, 65535
187         mov     edi, 1
188         call    __static_initialization_and_destruction_0(int, int)
189         pop     rbp
190         ret
```

Works Cited

"CS 2150 Final." *Quizlet*, quizlet.com/22777614/cs-2150-final-flash-cards/.

CS 2150: 09-Advanced-Cpp Slide Set, aaronbloomfield.github.io/pdr/slides/09-advanced-cpp.html#/5/11.

CS 2150: 09-Advanced-Cpp Slide Set, aaronbloomfield.github.io/pdr/slides/09-advanced-cpp.html#/5/2.

IBM Knowledge Center, www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.cbclx01/cplr380.htm.

"Multiple Inheritance in C++." *GeeksforGeeks*, 6 Sept. 2018, www.geeksforgeeks.org/multiple-inheritance-in-c/.

"Virtual Functions and Runtime Polymorphism in C++ | Set 1 (Introduction)." *GeeksforGeeks*, 5 Sept. 2018, www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/.

"Virtual Method Table." *Wikipedia*, Wikimedia Foundation, 16 Jan. 2019, en.wikipedia.org/wiki/Virtual_method_table.