

1 Inheritance

Consider the following example of simple inheritance in c++. It includes a derived class "Child" that inherits from the base class "Parent." It inherits Parent's x and y members and also has a public member of its own, "luckyFin."

```
1 class Parent
2 {
3     public:
4         void setX(int w)
5         {
6             x = w;
7         }
8         void setY(int h)
9         {
10            y = h;
11        }
12    protected:
13        int x;
14        int y;
15 };
16
17 class Child: public Parent
18 {
19     public:
20         int xPlusY()
21         {
22             return (x + y);
23         }
24
25         int luckyFin;
26 };
27
28 int main(void)
29 {
30     Child * nemo = new Child();
31
32     nemo->setX(2);
33     nemo->setY(3);
34
35     int add = nemo->xPlusY();
36
37     return 0;
38 }
```

The resulting assembly code is too long to list in full but snippets will be shown as the topics they are relevant to are discussed.

1.1 Initialization

The following x86 snippet shows the initialization of the nemo object.

```
1  call _Znwj
2  add  esp, 16
3  mov  DWORD PTR [eax], 0
4  mov  DWORD PTR [eax+4], 0
5  mov  DWORD PTR [eax+8], 0
6  mov  DWORD PTR [ebp-12], eax
7  mov  eax, DWORD PTR [ebp-12]
```

The call to `_Znwj` returns a pointer to the location of the newly created nemo object in memory. Then the data members of nemo are moved into memory at that location.

1.2 Data Layout

From the code above it is not immediately clear which element is stored at which location. This information can be deduced by considering a call to `setX` inherited from the Parent class. The assembly for such a call is as follows.

```
1  push 2
2  push eax
3  call _ZN6Parent4setXEi
```

with the function itself being

```
1 _ZN6Parent4setXEi:
2  push ebp
3  mov  ebp, esp
4  mov  eax, DWORD PTR [ebp+8]
5  mov  edx, DWORD PTR [ebp+12]
6  mov  DWORD PTR [eax], edx
7  pop  ebp
8  ret
```

Before the call is made, `eax` holds the address of the nemo object in memory. The `setX` method then moves the argument '2' located at `[ebp+12]` into `[eax]`, the lowest memory location in nemo. Similar analysis of the `setY` function shows us that `y` is located at `[eax+4]`. From this we can assume that 'luckyFin' is stored at `[eax+8]`.

The general rule that we can conclude from this examination is that within an object that inherits from a parent class the object's own member variables are stored first (at higher memory addresses) and then the inherited members are stored in reverse order with the first member of the parent class being stored at the lowest address.

1.3 Destruction

The first example will demonstrate what happens when a user created object goes out of scope. In order to more easily demonstrate this the c++ code has been modified in the following ways:

```
1 void scope() {
2     Child * nemo = new Child();
3     nemo->setX(2);
4     nemo->setY(3);
5     int add = nemo->xPlusY();
6 }
7
8 int main(void) {
9     scope();
10    return 0;
11 }
```

The last lines of assembly within the 'scope' function call look like this:

```
1 call _ZN5Child6xPlusYEv
2 add esp, 16
3 mov DWORD PTR [ebp-16], eax
4 leave
5 ret
```

As can be seen, the deallocation of the nemo object is very simple and operates just like deallocation of any other variables on the stack by adding to esp.

The second example will show what happens when something is 'destroyed' by the destructor.

```
1 class Planet {
2     public:
3         void setX(int w) {
4             x = w;
5         }
6         void setY(int h) {
7             y = h;
8         }
9     protected:
10        int x;
11        int y;
12 };
13
14 class DevourerOfWorlds {
15     public:
16        int luckyFin;
17        DevourerOfWorlds();
18        ~DevourerOfWorlds();
```

```
19
20     private:
21         Planet * earth;
22     };
23
24     DevourerOfWorlds::DevourerOfWorlds() {
25         earth = new Planet();
26     }
27
28     DevourerOfWorlds::~DevourerOfWorlds() {
29         delete earth;
30     }
31
32     void scope() {
33         DevourerOfWorlds galactus = DevourerOfWorlds();
34     }
35
36     int main(void) {
37         scope();
38         return 0;
39     }
```

Here the constructors and destructors have been explicitly defined. The destructor is automatically called when the program exits the scope of scope(). They compile to

```
1  _ZN16DevourerOfWorldsC2Ev:
2      push ebp
3      mov  ebp, esp
4      sub  esp, 8
5      sub  esp, 12
6      push 8
7      call _Znwj
8      add  esp, 16
9      mov  DWORD PTR [eax], 0
10     mov  DWORD PTR [eax+4], 0
11     mov  edx, DWORD PTR [ebp+8]
12     mov  DWORD PTR [edx+4], eax
13     leave
14     ret
15  _ZN16DevourerOfWorldsD2Ev:
16     push ebp
17     mov  ebp, esp
18     sub  esp, 8
19     mov  eax, DWORD PTR [ebp+8]
20     mov  eax, DWORD PTR [eax+4]
21     sub  esp, 12
22     push eax
23     call _ZdlPv
```

```
24     add    esp, 16
25     leave
26     ret
```

The constructor initializes galactus as we expect and also calls `_Znwj` to create the Planet earth object.

In the destructor, the address of earth is moved into `eax` which is then passed as an argument to `_ZdlPv` which deallocates the memory allocated to it. From there things proceed as normal with memory being deallocated by adding to `esp` and then `leave` and `ret` making up the standard epilogue.