x86 Disassembly/Objects and Classes

*The Objects and Classes page of the **X86 Disassembly** Wikibook is a stub. You can help by expanding this section.*

Object-Oriented Programming

Object-Oriented (OO) programming provides for us a new unit of program structure to contend with: the **Object**. This chapter will look at disassembled classes from C++. This chapter will not deal directly with COM, but it will work to set a lot of the groundwork for future discussions in reversing COM components (Windows users only).

Classes

A basic class that has not inherited anything can be broken into two parts, the variables and the methods. The non-static variables are shoved into a simple data structure while the methods are compiled and called like every other function.

When you start adding in inheritance and polymorphism, things get a little more complicated. For the purposes of simplicity, the structure of an object will be described in terms of having no inheritance. At the end, however, inheritance and polymorphism will be covered.

Variables

All static variables defined in a class resides in the static region of memory for the entire duration of the application. Every other variable defined in the class is placed into a data structure known as an object. Typically when the constructor is called, the variables are placed into the object in sequential order, see **Figure 1**.

A:

```
class ABC123 {
public:
    int a, b, c;
    ABC123():a(1), b(2), c(3) {};
};
```

B:

```
0x00200000  dd  1   ;int a
0x00200004  dd  2   ;int b
0x00200008  dd  3   ;int c
```

Figure 1: An example of what an object looks like in memory
Figure 1.A: The definition for the class "ABC123." This class has three integers, a, b, and c. The constructor sets 'a' to equal 1, 'b' to equal 2, and 'c' to equal 3.
Figure 1.B: How the object ABC123 might be placed in memory, ordering the variables from the class sequentially. At memory address 0x00200000 there is a double word integer (32 bits) with a value of 1, representing the variable 'a'. Memory address 0x00200004 has a double word integer with the value of 2, representing the variable 'b'. And at memory address 0x00200008 there is a double word integer with a value of 3, representing the variable 'c'.

However, the compiler typically needs the variables to be separated into sizes that are multiples of a word (2 bytes) in order to locate them. Not all variables fit this requirement, namely char arrays; some unused bits might be used pad the variables so they meet this size requirement. This is illustrated in **Figure 2**.

A:

```
class ABC123{
public:
    int a;
    char b[3];
    double c;

    ABC123():a(1),c(3) { strcpy(b,"02"); };
};
```

B:

```
0x00200000  dd  1       ;int a              ; offset = abc123 + 0*word_size
0x00200004  db  '0'     ;b[0] = '0'         ; offset = abc123 + 2*word_size
0x00200005  db  '2'     ;b[1] = '2'
0x00200006  db  0       ;b[2] = null
0x00200007  db  0       ;<= UNUSED BYTE
0x00200008  dd  0x00000000  ;double c,  lower 32 bits   ; offset = abc123 + 4*word_size
0x0020000C  dd  0x40080000  ;double c,  upper 32 bits
```

Figure 2: An example of an object having a padded variable
Figure 2.A: A new definition for the class "ABC123." This class has one 32 bit integer, a. One 3 byte char array, b. And one 64 bit double, c. The constrictor sets 'a' to 1, 'b' to "02", and 'c' to 3.
Figure 2.B Shows how ABC123 might be stored in memory. The first double word in the object is the variable 'a' at location 0x00200000 with a value of 1. Variable 'b' starts at the memory location 0x00200004. It's three bytes containing three chars, '0','2', and the null value. The next available address, 0x00200007, is unused since it's not a multiple of a word. The last variable 'c', starts at 0x00200008 and it two double words (64 bits). It contains the value 3.

In order for the application to access one of these object variables, an object pointer needs to be offset to find the desired variable. The offset of every variable is known by the compiler and written into the object code wherever it's needed. **Figure 3** shows how to offset a pointer to retrieve variables.

```
;abc123 = pointer to object
mov eax, [abc123]   ;eax = &a   ;offset = abc123+0*word_size    = abc123
mov ebx, [abc123+4] ;ebx = &b   ;offset = abc123+2*word_size    = abc123+4
mov ecx, [abc123+8] ;ecx = &c   ;offset = abc123+4*word_size    = abc123+8
```

Figure 3: This shows how to offset a pointer to retrieve variables. The first line places the address of variable 'a' into eax. The second line places the address of variable 'b' into ebx. And the last line places the variable 'c' into ecx.

Methods

At a low level, there is almost no difference between a function and a method. When decompiling, it can sometimes be hard to tell a difference between the two. They both reside in the text memory space, and both are called the same way. An example of how a method is called can be seen in **Figure 4**.

A:

```
//method call
abc123->foo(1, 2, 3);
```

B:

```
push 3          ; int c
push 2          ; int b
push 1          ; int a
push [ebp-4]        ; the address of the object
call 0x00434125 ; call to method
```

Figure 4: A method call.
Figure 4.A: A method call in the C++ syntax. abc123 is a pointer to an object that has a method, foo(). foo() is taking three integer arguments, 1, 2, and 3.
Figure 4.B The same method call in x86 assembly. It takes four arguments, the address of the object and three integers. The pointer to the object is at ebp-4 and the method is at address 0x00434125.

A notable characteristic in a method call is the address of the object being passed in as an argument. This, however, is not a always a good indicator. **Figure 5** shows function with the first argument being an object passed in by reference. The result is function that looks identical to a method call.

A:

```
//function call
foo(abc123, 1, 2, 3);
```

B:

```
push 3          ; int c
push 2          ; int b
push 1          ; int a
push [ebp+4]        ; the address of the object
call 0x00498372 ; call to function
```

Figure 5: A function call.
Figure 5.A: A function call in the C++ syntax. foo() is taking four arguments, one pointer and three integer arguments.
Figure 5.B: The same function call in x86 assembly. It takes four arguments, the address of the object and three integers. The pointer to the object is at ebp-4 and the method is at address 0x00498372.

Inheritance & Polymorphism

Inheritance and polymorphism completely changes the structure of a class, the object no longer contains just variables, they also contain pointers to the inherited methods. This is due to the fact that polymorphism requires the address of a method or inner object to be figured out at runtime.

Take **Figure 6** into consideration. How does the application know to call D::one or C::one? The answer is that the compiler figures out a convention in which to order variables and method pointers inside the object such that when they're referenced, the offsets are the same for any object that has inherited its methods and variables.

```
A *obj[2];
obj[0] = new C();
obj[1] = new D();

for(int i=0; i<2; i++)
    obj[i]->one();
```

Figure 6: A small C++ polymorphic loop that calls a function, one. The classes C and D both inherit an abstract class, A. The class A, for this code to work, must have a virtual method with the name, "one."

The abstract class A acts as a blueprint for the compiler, defining an expected structure for any class that inherits it. Every variable defined in class A and every virtual method defined in A will have the exact same offset for any of its children. **Figure 7** declares a possible inheritance scheme as well as it structure in memory. Notice how the offset to C::one is the same as D::one, and the offset to C's copy of A::a is the same as D's copy. In this, our polymorphic loop can just iterate through the array of pointers and know exactly where to find each method.

A:

```
class A{
public:
    int a;
    virtual void one() = 0;
};
class B{
public:
    int b;
    int c;
    virtual void two() = 0;
```

```
};
class C: public A{
public:
    int d;
    void one();
};
class D: public A, public B{
public:
    int e;
    void one();
    void two();
};
```

B:

```
;Object C
0x00200000  dd  0x00423848  ; address of C::one ;offset = 0*word_size
0x00200004  dd  1       ; C's copy of A::a  ;offset = 2*word_size
0x00200008  dd  4       ; C::d          ;offset = 4*word_size

;Object D
0x00200100  dd  0x00412348  ; address of D::one ;offset = 0*word_size
0x00200104  dd  1       ; D's copy of A::a  ;offset = 2*word_size
0x00200108  dd  0x00431255  ; address of D::two ;offset = 4*word_size
0x0020010C  dd  2       ; D's copy of B::b  ;offset = 6*word_size
0x00200110  dd  3       ; D's copy of B::c  ;offset = 8*word_size
0x00200114  dd  5       ; D::e          ;offset = 10*word_size
```

Figure 7: A polymorphic inheritance scheme.
Figure 7.A defines the inheritance scheme. It shows that class C inherits class A, and class D inherits class A and class B.

Figure 7.B shows how the inheritance scheme might be structured in memory. Class C's object has everything that was declared in class A in the first two double words. The remainder of the object was defined by class C. Class D's object also has everything that was declared in class A in the first two double words. Then the next three double words is everything declared in class B. And the last double word is the variable defined by class D.

Classes Vs. Structs