SI413 Unit 11: OOP Implementation

Chris Brown (wcbrown@usna.edu)

Fall 2023

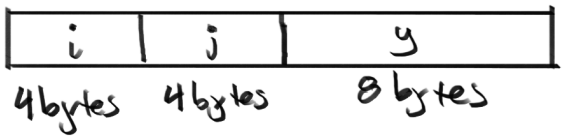
Object oriented programming (OOP) has four central ideas: encapsulation, data hiding, inheritance and polymorphism. We are doing to focus on encapsulation, inheritance and polymorphism, and specifically focus on how they are implemented in languages like C++ and Java. Remember that an "object" in OOP is a collection into a single package of related pieces of *data* and *functions* operating on that data.

1 Structs in languages like C++ and Java

First we need to discuss how data gets packaged together in languages like C, C++ and Java. Let's just look at a simple struct/class definition like this:

```
class Foo
{
    int i;
    int j;
    double y;
};
```

In this case we need at least 16 bytes, because each int takes 4 bytes and the double takes 8. The bytes for the fields are typically laid out sequentially, in the order fields appear textually in the class definition. So for Foo, the first four bytes are i, the next four are j, and the last eight are y.



So the compiler/interpreter will keep track of the *offset* of each field in the struct/class. If you access a field, the compiler will know its type (and thus how many bytes) and its offset within the object, so it can compute exactly which bytes belong to that object. The example below shows how the compiler really transforms field access within structs to nothing more than pointer arithmetic with offsets and some type casting.

```
class Foo          // Offset
{
  int i;           //  0
  int j;           //  4
  double y;        //  8
};

int main() {
  // Example code     (as the compiler translates it)
  Foo f;           // f = (16 bytes of space on the stack);
  f.i = 7;         // *(int*)(&f + 0) = 7;
  f.j = 42;        // *(int*)(&f + 4) = 42;
```

```
  f.y = 0.357       // *(double*)(&f + 8) = 0.357;
}
```

If you don't believe me, you can try this little experiment. Compile and run the code below, and you will see that the fields are assigned their proper values using the "pointer to f + offset, cast to the proper type" method:

```
#include <iostream>
using namespace std;

class Foo
{
public:
  int i,j;
  double y;
};

int main()
{
  Foo f;

  // Assign values to fields as the compiler does it
  // Note, the cast to unsigned char* is to treat f as a raw pointer
  *(int*)((unsigned char*)&f + 0) = 7;
  *(int*)((unsigned char*)&f + 4) = 42;
  *(double*)((unsigned char*)&f + 8) = 0.357;

  // Print out letting the compiler do the hard stuff
  cout << f.i << endl;
  cout << f.j << endl;
  cout << f.y << endl;
  return 0;
}
```

1.1 A quick note on the reality of the placement of fields

For efficiency reasons, compilers typically prefer to align N-byte objects to start at memory locations with addresses that are zero mod N. For example, doubles start at addresses that (in hex!) end in either zero or eight, whereas ints start at addresses that (in hex!) end in 0, 4, 8 or C (note that C=12 in hex). This means that the layout of fields in objects might leave gaps of unused memory.

For example, consider this code, where the struct now has only one int and one double field:

```
#include <iostream>
using namespace std;

class Foo
{
public:
  int i;
  double y;
};

int main()
{
```

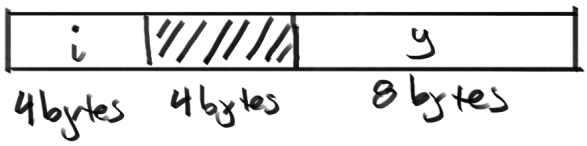
```
  Foo f;
  cout << "sizeof(f) = "
       << sizeof(f) << endl
       << &f << " f" << endl
       << &f.i << " f.i" << endl
       << &f.y << " f.y" << endl;
  return 0;
}
```

Here is what the output might look like when we compile and run it:

```
sizeof(f) = 16
0x7ffc208ee160 f
0x7ffc208ee160 f.i
0x7ffc208ee168 f.y
```

- First notice that the size of a Foo object is 16 bytes even though one 4-byte int and one 8-byte double should only take 12 bytes.
- Second notice that the the address shows that the int i occupies the first four bytes of the Foo object (the address of f and f.i are the same), and the double y occupies the final eight bytes. This leaves a 4-byte gap between them!
- This gap occurs because the compiler wants to align the 8-byte double on 8-byte boundaries — i.e. it wants addresses ending in either 0 or 8. That forces a gap.

In other words, the struct layout now looks like this:



Of course these issues of alignment are not usually something we think much about, but when you want to be as efficient as possible with memory, you might have to take it into account!

2 Methods (aka member functions): part 1, the non-polymophic case

In this section we'll examine how methods (aka member functions) are implemented in languages like C++ and Java in the case in which there is no polymorphism. Suppose we have something like this:

```
class Foo {
  public int i;
  public int j;
  public double y;

  public int bar() {
    i++;
    return j;
  }
}
```

...where there is no inheritance and thus no polymorphism to worry about. The method "bar" is different from normal functions because it has access to the fields of the object on which it was called. For example, if

we have Foo f and call f.bar(), the "bar" function will access f's i and j fields. We handle this by rewriting methods to add a pointer to the object on which the method was called as the first parameter to the function.

If translating the above to C (to see how it would get compiled), the bar method might be written like:

```
int bar(Foo* this) {
  this->i++;
  return this->j;
}
```

or actually, taking account what we understand about how structs work, this function definition would be compiled to:

```
int bar(Foo* this) {
  *(int*)((unsigned char*)this + 0)++;       // offset 0 for i
  return *(int*)((unsigned char*)this + 4); // offset 4 for j
}
```

Additionally, each method call would have to be rewritten as well, so that for example f.bar() is rewritten as bar(f)

In this way methods (aka member functions) and method calls are nothing more than syntactic sugar: we can systematically rewrite them as normal function definitions and normal function calls. At least for "final" methods in Java or non-virtual member functions in C++ (i.e. when there is no polymorphism) this is how things work.

If you look at rewritten version of bar above, you see that bar picks out the fields of the object based solely on offsets from the "this" pointer. That's important!

3 Inheritance

The fundamental problem with inheritance is that code that works with objects of the base class should also work with objects of the derived class types. Let's continue with the class Foo from the previous section and derive from it a new class Foov2

```
class Foov2 extends Foo {
  public boolean tag;
  public void rat() {
    // print out the value of y and tag with a space between
    System.out.format("%g␣%b%n", y, tag);
  }
};
```

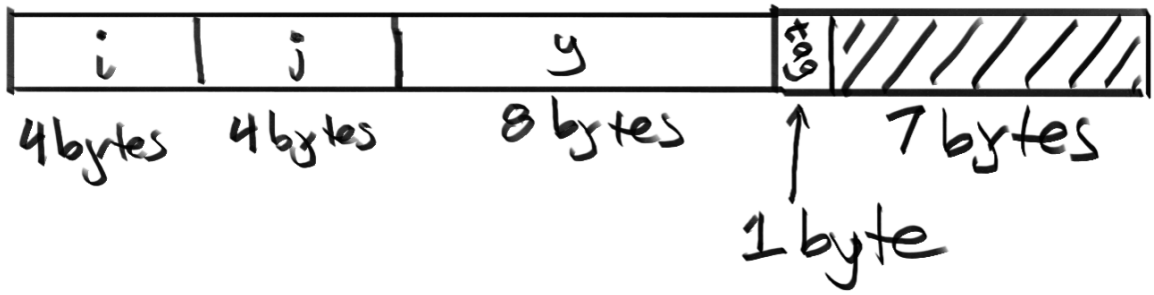
...and let's suppose that we instantiate a Foov2 object and set some of its fields and call some of its methods

```
Foov2 f2 = new Foov2();
f2.i = 42;
f2.j = 23;                    // <—— How does any of this work?!?!?
f2.y = 3.5;
f2.tag = true;
System.out.println(f2.bar());
f2.rat();
```

Let's do a deep dive and see how this might actually work.

- **How are the fields in an object of the derived class laid out?**

The layout of the derived class is simple: the compiler *concatenates* the object layout of the base class with the layout for the new fields of the derived class. That means that field "bool tag;" will just be stuck on the end of the i,j,y fields, which are laid out just as they are for class Foo on its own.



We know that getting/setting fields is really just done by offsets from the this pointer, and now we know that those offsets are enchanged for the fields inherited from the base class.

- **How do calls of methods from the base class work on objects of the derived class?**

  Let's consider the method bar() from the base class. As we saw, that method definition is actually syntactic sugar. It is compiled as something like this (in C syntax):

```
int bar(unsigned char* this) {
    int* iptr = (int*)(ptr + 0); // offset 0
    int* jptr = (int*)(ptr + 4); // offset 4
    *iptr++;
    return *jptr;
}
```

  Now if we execute this code with a pointer to a Foov2 object as its argument, like bar(&f2), what happens? Well: f2−>i sits at offset 0 and f2−>j sits at offset 4, so ... the function still works perfectly. It is blissfully unaware that at offset 16 from &f2 there is more data that pure Foo objects wouldn't have. It never looks beyond the offsets of fields in Foo objects.

  So inherited methods simply work without any extra shennanigans purely as a by-product of the way new fields of the derived class are contatenated onto the field of the base class.

- **How do new methods in the derived class work?**

  So how do new methods of the derived class work? Hopefully the preceeding discussion makes it clear. They work just like the methods from the base class work, except that they may reference fields from the derived class (i.e. offsets beyond the area of memory in which the fields of the base class live).

4 Methods (aka member functions): part 2, polymorphic functions!

So now we get to the last issue: how do polymorphic function calls work? C++ and Java handle polymorphic calls similarly from an implementation perspective. From a programmer's perspective, an important difference is that functions are polymorphic by default in Java (but you can turn it off by marking a method "final"), but in C++ functions are by default *not* polymorphic (though you can turn it on by marking a method "virtual"). So let's make the function bar() in class Foo polymorphic, and let's override it in class Foov2. We'll set it up so that if there are no command-line arguments, pointer p points to a Foo object, and if there are command-line arguments pointer p points to a Foov2 object.

We're doing this in C++ now so we can actually look at the addresses at run-time and see how it works, but it will be the same if we compiled from java. Here's the source code:

```
#include <iostream>
using namespace std;
```

```
class Foo
{
public:
  int i,j;
  double y;
  virtual int bar() { return 23; }
};

class Foov2 : public Foo
{
public:
  bool tag;
  int bar() { return 42; }
};

int offset(void* p, void* q) { return (unsigned char*)q - (unsigned char*)p; }

int main(int argc, char** argv)
{
  Foo f1;
  Foov2 f2;
  Foo* p = argc == 1 ? &f1 : &f2;
  cout << "p->bar() = " << p->bar() << endl;
  //-----------------------^^^^^^^----------Polymorphic call
  cout << "sizeof(*p) = "
       << sizeof(*p) << endl
       << p << " p" << endl
       << &p->i << " p->i" << "   offset " << offset(p,&p->i) << endl
       << &p->j << " p->i" << "   offset " << offset(p,&p->j) << endl
       << &p->y << " p->y" << "   offset " << offset(p,&p->y) << endl;
  return 0;
}
```

So now we'll run this program twice, once with no command-line arguments and once with command-line arguments.

No command-line args:

```
$ ./a.out
p->bar() = 23
sizeof(*p) = 24
0x7fffbab96040 p
0x7fffbab96048 p->i    offset 8
0x7fffbab9604c p->i    offset 12
0x7fffbab96050 p->y    offset 16
```

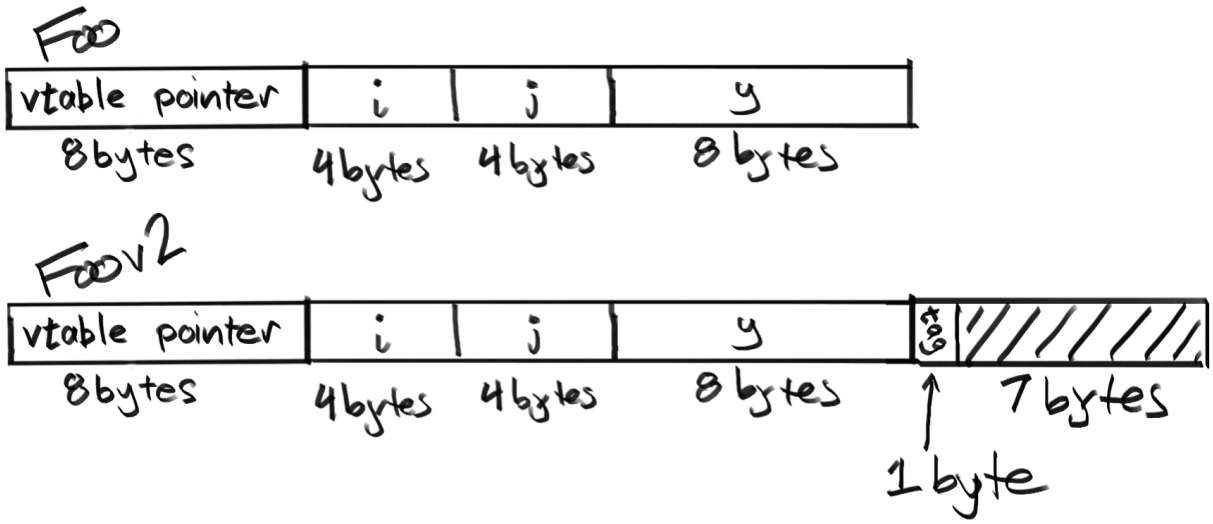
With command-line args:

```
$ ./a.out dummy
p->bar() = 42
sizeof(*p) = 24
0x7ffe61eea620 p
0x7ffe61eea628 p->i    offset 8
0x7ffe61eea62c p->i    offset 12
0x7ffe61eea630 p->y    offset 16
```

This is the prototypical polymorphic function call. We have one call site in the code – p->bar() – and sometimes that call site results in Foo's bar() function getting called, and other times it results in Foov2's bar() function getting called. That decision, note, is a *runtime* decision. The compiler doesn't know which version of bar() will be called, and it can change from run to run (or in other examples where we have loops, it can change within a single run of the program).

Important: The secret to how this is implemented hinted at by the other output of this program. Did you notice that the offset of field i, the first field in the Foo object, jumped? It went from offset 0 to offset 8. Why? And what do you think the compiler is doing with those first eight bytes in a Foo object?

Because we have marked bar() as "virtual" in class Foo, the compiler has to handle polymorphic function calls, and the extra eight bytes that have been inserted into the front of Foo objects is integral to how this is done. Those first eight bytes comprise a pointer to a data structure called the "vtable" for the class.



If we were to print the value of the vtable pointer for any instance of Foo, it would always be the same. So the vtable is shared by all instances of the class. The vtable pointer for Foov2 instances are all the same as one another, but different from the Foo vtable pointers. So in some sense, the vtable pointer is what makes Foo instances different from Foov2 instances. The vtable contains function pointers for *methods* at fixed offsets just as the instance contains values for data *fields* at fixed offsets. When we make the call p->bar() here's what happens:

1. the first 8 bytes of the memory pointed to by p is read (this is the vtable pointer)
2. we read the value of the function pointer for our function call from the vtable pointer plus some fixed offset that is unique to bar
3. we call the function given by that function pointer

So suppose that the fixed offset associated with bar() is 24 bytes. We will read the function pointer at address vtablepointer + 24. Since vtablepointer is different for Foo's than for Foov2's, we get a different function address for Foo objects than for Foov2 objects.

Here's one last demo which, I hope, will drive home this point.

```
#include <iostream>
using namespace std;

class Foo
{
public:
  int i,j;
  double y;
```

```
  virtual int bar() { return 23; }
};

class Foov2 : public Foo
{
public:
  bool tag;
  int bar() { return 42; }
};

int main(int argc, char** argv)
{
  Foo A;
  Foov2 X;
  Foo* ptr1 = &A;
  Foo* ptr2 = &X;

  // polymorphic calls to bar() for A then X
  cout << "ptr1->bar() = " << ptr1->bar() << endl;
  cout << "ptr2->bar() = " << ptr2->bar() << endl;

  // swap the first 8 bytes of A and first 8 bytes of X
  unsigned char* pA = (unsigned char*)&A;
  unsigned char* pX = (unsigned char*)&X;
  for(int i = 0; i < 8; i++)
    swap(pX[i],pA[i]);

  // polymorphic calls to bar() for A then X
  cout << "ptr1->bar() = " << ptr1->bar() << endl;
  cout << "ptr2->bar() = " << ptr2->bar() << endl;

  return 0;
}
```

And here is what happens when we run it:

```
$ g++ ex4.cpp
$ ./a.out
ptr1->bar() = 23
ptr2->bar() = 42
ptr1->bar() = 42
ptr2->bar() = 23
```

We have ptr1 that points to Foo A, and ptr2 that points to Foov2 X. We call ptr1->bar() followed by ptr2->bar() and see, as we expect, 23 followed by 42. But then we swap the first eight bytes of Foo A and Foov2 X. Now they point to each other's vtables. When we call ptr1->bar() followed by ptr2->bar() again, we see that Foo A is now calling the Foov2 bar() implementation and Foov2 X is now calling the Foo bar() implementation. So by changing the vtable pointer in the instances, we are changing how those polymorphic calls act. BTW: don't ever do something like this in a real program!