Removing Classes from a Program

16th Sep 2014

We learn how to store a class and its member fields/methods at an assembly code / C code level. Some assumptions made:

1. int occupies 4 bytes in memory
2. Any memory location can be accessed using 4 bytes (equivalently, size of a pointer is 4 bytes).

A class in Java contains both fields and methods. We look at how to store each of these, first for a simple class, and then for classes with inheritance.

Simple Classes

Consider the following class definition:

```
class B {
    int x;
    int y;
    int f1 (int z) {
        int t1;
        t1 = x + y + z;
        return t1;
    }
    int f2 () {
        return x;
    }
    int f3 () {
        //some code
    }
    int f4 () {
        //some other code
    }
}
```

Converting Methods: Let us see how to convert a method in a class to a simple function. Take the example of B::f1.

```
int f1 (int z) {
    int t1;
    t1 = x + y + z;
    return t1;
}
```

The x and y are fields in the class, and hence, we need to pass references to them. We do so by passing an extra parameter to the function:

```
int B_f1 (B mthis, int z) {
    int t1;
    t1 = mthis.x + mthis.y + z;
    return t1;
}
```

Now, the function can stand alone - it no longer needs to be called from a class.

Storing Fields: Let us look at how we could store fields of a class. An idea would be to simply store all fields in contiguous memory locations. Hence, to access any field, we can just give an offset, which can be statically determined with respect to a class. However, this is not enough, as we also need to know which methods belong to a particular class. We can store these as function pointers, in a VTable (virtual table). For example, for class B, we would store the following:

VTABLE
x
y

Pointer to B_f1
Pointer to B_f2
Pointer to B_f3
Pointer to B_f4

where VTABLE points to the array containing the methods of class B. Hence, fields of B can be accessed by giving offsets.

Translating Expressions and Function calls: For example, let us look at the implementation of
B b = new B();

```
x1 = allocate (12);      //we need to allocate memory for storage
x1[4] = 0;               //Initialise the variable at an offset of 4 (x)
x1[8] = 0;               //Initialise the variable at an offset of 8 (y)
x2 = allocate (16);      //Allocate memory for the function pointers
x2[0] = & B_f1;          //First function pointer points to B_f1
x2[4] = & B_f2;          //Second function pointer points to B_f2
x2[8] = & B_f3;          //Third function pointer points to B_f3
x2[12] = & B_f4;         //Fourth function pointer points to B_f4
x1[0] = x2;              //Set up pointer to the VTABLE
```

This translates the expression new B(). To initialise b, we simply set
b = x1;
To translate function calls, we can do something similar. Consider the function call
int z = b.f1(5);
We handle this as follows:

```
t1 = b[0];           //Get address of the VTABLE
z = t1[0](b, 5);     //t1[0] points to B_f1
                     //We need to pass both the memory locations of the class
                     // as well as the function parameters
```

Similarly, for the function call
int z = b.f4(6);
we would call

```
t1 = b[0];              //Get address of the VTABLE
z = t1[12](b, 6);       //t1[12] points to B_f4
```

Classes with Inheritance

Let class A extend class B as follows:

```
class A extends B {
    int x;
    int z;
    int f1 (int p) {
        int t2 = f2();
        int t1 = 2*x + y - p + t2;
        return t1;
    }
    int set (int a, int b) {
        x = a;
        z = b;
    }
}
```

Notice that function f1() overrides the function in B. We construct the function for storage and the VTABLE as follows:

VTABLE
B_x
B_y
A_x
A_z

Pointer to A_f1
Pointer to B_f2
Pointer to B_f3
Pointer to B_f4
Pointer to A_set

Note that A_f1 replaces B_f1 in the VTABLE as A_f1 overrides B_f1. Also note that we store the parent class' fields/ methods first. This ensures that the structures for the parent class and the child class have the same offset for the variables of the parent class.

Converting Methods: Let us translate the method f1() in class A

```
int f1 (int p) {
    int t2 = f2();
    int t1 = 2*x + y - p + t2;
    return t1;
}
```

Referring to B's methods, we see that f2() returns value of B::x. This is achieved through the following translation:

```
int B_f2 (B bthis) {
    return bthis[4];
}
```

To convert the function f1(), we add the object calling the function to the parameters list, and get:

```
int A_f1 (A athis, int p) {
    int t2 = athis[0][4](athis);
    int t1 = 2*athis[12] + athis[8] - p;
    return t1;
}
```

The function call to f2() automatically returns B::x, as the offset of B::x is the same in both the parent and child class. Notice that x refers to the field in class A, whereas y refers to the field in class B. Offsets are provided accordingly.

Translating Statements and Function Calls: Consider the following set of statements.

```
A a = new A();
B b = a;
int p = a.f1(5);
print(a.x);
print(b.x);
```

Allocating memory is the same as that described for class B.
a.f1(5);
is translated as follows:

```
t1 = a[0];              //Get address of the VTABLE
p = t1[0](a, 6);        //t1[0] points to A_f1
```

Now, consider the print statements.
print(a.x); The x refers to the field in class A. Hence, this would become

```
print(a[12]);           //A_x is stored at an offset 12
```

However, for print(b.x);, as b is of static type B, b.x refers to the field in class B, and is translated to

```
print(b[4]);            //B_x is stored at an offset 4
                        // for both parent and child class
```

Notice that this would remain the same even if b was not dynamically type-casted to A.