Supporting Class / C++
Lecture Notes

Goal

You started with an understanding of how to write Java programs. This course is about explaining the path from Java to executing programs. We proceeded in a mostly bottom-up manner: we started down at the hardware, and have worked out way up through compiling, linking, and executing C programs.

C is at least one step away from Java. One of the major distinctions is that Java supports classes, while C does not.

Our goal in this section is understand how classes are supported. Instead of stepping all the way to Java, we're stepping only to C++. C++ is basically C + classes. (Java is C + classes + some other things, e.g., memory safety.)

Preliminaries

We assume as background an understanding of how C programs are compiled and linked. That material is covered in the book (as well as in lectures).

The goal of this material is to look at how to compile programs written in languages that support classes and inheritance. We're most interested in things that are different than when compiling C.

There are three players in the implementation: the compiler, the linker, and the runtime system (a set of functions loaded with the code you wrote that support its basic operation). They work together to implement execution of your code.

"Static" means done without actually running the code. "Dynamic" means done while running the code. The primary distinction between the two is that dynamic has access to the particular path through the code that has been followed (for example, what code called the method we're currently executing), while static doesn't. Reading a code file is a kind of static analysis of it. Running the debugger on your code, setting a breakpoint, and examining variable values while at the breakpoint is an example of dynamic analysis.

We prefer to implement a language feature (e.g., classes) statically, rather than dynamically. That's more efficient, because we compile once (ideally), but we run the program repeatedly. Not everything can be determined statically, though, as we'll see. Those things that can't be determined statically must be implemented, at least in part, by code that runs with your code, i.e., dynamically. (We've already seen this distinction in compiling C. Simple statements can be turned into machine code rather directly. Code for subroutines must follow a subroutine call convention – that is, there is a bunch of code that supports subroutine call/return. That code is injected directly into the caller and called methods, but we can think of it as runtime support for the procedure call language feature.)

I'll be using the code samples from these lectures, without retyping that code here:

> http://www.cs.washington.edu/education/courses/cse351/11wi/lectures/cppExample.pdf
>
> http://www.cs.washington.edu/education/courses/cse351/11wi/lectures/cppExample.tar.gz

Finally, you should think of the information here as giving you the essential idea of how classes are implemented. The details of how to implement is left to each individual compiler, and so there's no "one right way." Additionally, there are many (many) details that an actual compiler has to be

concerned with, but just get in the way here, and so aren't considered.

Naming

C defines rules about the scopes of names – where in the code you can use the name. C++ has similar scoping rules. In addition, C++ introduces the notions of private, protected, and public, which are just like those ideas in Java. Enforcement of them is done statically in C++. When some code implementing class `foo` tries to access a private instance variable of an object of class `bar`, the compiler raises an error. It should be easy to see that the compiler has enough information at compile time to reliably apply scope rules.

Unlike C, though, the names in a C++ program form a hierarchical name space (as they do in Java). `foo::read()` is different from `bar::read()`, for instance, meaning that the name 'read' may not uniquely name anything. This is unlike C, where there is a flat, global name space, and the language allows only a single instance of any name to exist. (You can have only one method named `read()` in C.)

Making matters worse, C++ allows polymorphic functions. For instance, there may be a method of class `foo` with type signature `int read()` and another with signature `int read(char* fName)`. So, even the name `foo::read` isn't guaranteed to be unique.

A final complication is that these names have to be given to the linker, so that it can perform final linking of the code. Depending on the linker implementation, it may support only a flat name space (not a hierarchical one).

To resolve all these issues, the hierarchical names in C++ are "mangled" to form flat names. As one example, taken from the code sample given for these lectures, the method `int read()` in class `Polynomial` has mangled name `_ZN10Polynomial4readEv`. Clearly the simple class and method names form part of the mangled name. The additional characters encode things like the types of arguments, so that two different `Polynomial::read` functions (which is allowed by the language so long as the type signatures are sufficiently distinct) have distinct mangled names.

Implementing Objects: Simple Classes

Have a look at `Vector.h`, which defines the `Vector` class. It shouldn't be much of a stretch to see that the state of an object of type `Vector` can be represented by a `struct` whose fields correspond to the instance variables of the `Vector` class[1]:

```
typedef struct {
      int capacity;
      int length;
      double* pVec;
} VectorStruct;
```

Creating an object of type `Vector` (e.g., `pVec = new Vector();`) is now doing

```
pVec = (VectorStruct*)malloc(sizeof(VectorStruct));
pVec->Vector();  // invoke the constructor code on the newly allocated space
```

That second line brings up the small point of how what "invoking a method on an object means," in terms of assembler/hardware instructions. The hardware doesn't support the notion of object, just of procedure.

---

1 None of the "code" in this wrteup is guaranteed to actually compile. C++ sometimes/often requires syntax whose inclusion won't make anything clearer, so I'm not trying to be faithful to its demands.

The way it's done is to make the pointer to the object an implied first argument to all class methods, so that method invocation on an object becomes just procedure invocation where "the object" is an argument. The actual invocation of the constructor would therefore be something like this[2]:

```
Vector(pVec);
```

You write `pVec->Vector()` in C++, because that syntax expresses the object-y idea, but at execution the pointer to the object (`struct`) is just an argument to the procedure code.

Making the object reference the first argument works because the compiler does that uniformly – it does it when compiling invocations, and it does it when compiling the code for the method. There's no question of whether the object reference has been included as the first argument for any particular call, because the compiler makes sure it always is. Similarly, there's no question if the method's code expects an object reference as the first argument, because the compiler makes sure it always does. It's therefore possible, and correct, to compile the invocations even when you can't see the method's code (because they're in some other file), and to compile the method even when you can't see the calling code.

> *Note, by the way, that if someone forces you to use C, which doesn't have classes, you can (and maybe should) adopt a style that mimics the above, to make C more object-y: you explicitly make the first argument of a set of methods a reference to a struct, and put all those methods in a single file (with a name like Vector.c). Voila, a Vector class in C, sort of.*

Now suppose some code does something like this:

```
int x = pVec->capacity;
```

If that line of code isn't in a method that implements the `Vector` class, the compiler raises an error, because capacity is a private instance variable. (Note that the compiler always has enough information to determine this at compile time.) If that line is legal in the language, then it is compiled using the understanding that pVec points at a `struct`, and that field capacity is at offset 0 in that `struct`. (At the hardware level, if pVec is in register 2, we address capacity as $0(r2).)

Implementing Objects: Inheritance and Instance Variables

Suppose we class `bar` is a subclass of class `foo`, and the two have instance variables as indicated by the partial class definitions below:

```
class foo {
      private:
            int x;
      public:
            int y;
};
class bar : public foo {
      public:
            int z;
};
```

Then the state of an object of type `foo` is stored in a `struct` with two fields, `x` and `y`. The state of an object of type `bar` is a `struct` with three fields, `x`, `y`, and `z`. The `bar struct` needs the field `x`, even though it is private to class `foo`, because an object of type `bar` can be given to a method expecting a `foo`. For instance, consider this static method:

---

2 This is only the first part of what is actually required, in general. We'll see the second part shortly. It, on its own, works for this invocation though.

```
int uselessMethod( foo* pF) {
      pF->y = 0;
      return 0;
}
```

If I have a pointer to a `bar`, say `pB`, I can invoke `uselessMethod(pB)`.

In general, the `struct` that holds the state of an object of class C is formed by concatenating all the instance variables, starting at the base class and working down through the class hierarchy until reaching C.

Using this scheme, it's possible to compile `uselessMethod` even though we don't know at compile type the exact type of the object `pF` points at. What we do know is that it's "at least" a `foo`; it might be some subclass of a `foo`, though. Because of the concatenation in forming the `structs`, the state of every subclass object will be represented by a `struct` whose first elements match the elements in a `Foo struct`. That means that `pF->y` always means "4 bytes past the beginning of the `struct`," no matter what the specific type of the object is that `pF` points at. Therefore, the compiler can generate code for `uselessMethod` even though it has no way to know exactly what the type of argument will be.

Summarizing, the state of a `foo` object is represented by this `struct`:

```
struct {
      int x;
      int y;
}
```

The state of a `bar` is represented by this `struct`:

```
struct {
      int x;
      int y;
      int z;
}
```

At some future date, someone might subclass `bar`, and pass an object of this new subclass to `uselessMethod`. The states of objects of this future subclass might be saved in a `struct` like this:

```
struct {

      int x;
      int y;
      int z;
      int a;
      int b;
}
```

The point is, `uselessMethod` is expecting a `foo`. Class `foo` promises the first two elements of the `struct` (`x` and `y`), and nothing more. Every subclass will have `x` and `y` as the first two elements. They might also have more, but `uselessMethod` doesn't care – all it can operate on are the instance variables offered by something of the declared class (`foo`). Even if you, the programmer, somehow knew that your program always passed a pointer to a `bar` to `uselessMethod`, the line `pF->z = 2;` wouldn't compile (if put in `uselessMethod`), because `z` isn't an instance variable of type `foo`.

Implementing Objects: Virtual Method Invocation

Now lets look at the situation for methods in classes. Using `foo` and its subclass `bar` again, suppose we have this[3]:

3 This code actually compiles: g++ example.cpp

```
#include <stdio.h>
class foo {
   public:
      void sfunc() { printf("foo\n"); }
      virtual void vfunc()  { printf("foo\n"); }
};
class bar : public foo {
   public:
      void sfunc() { printf ("bar\n"); }
      void vfunc() { printf ("bar\n"); }
      virtual void anotherVfunc(){ printf("bar\n"); }
};
int main(int argc, char* argv[]) {
      bar* pB = new bar();
      foo* pF = pB;

      pF->vfunc();  // prints 'bar'
      pF->sfunc();  // prints 'foo'
      pB->sfunc();  // prints 'bar'

      return 0;
}
```

The `virtual` in the declaration of `vfunc()` in `foo` means that subclasses can override the method. The lack of `virtual` in the declaration of `sfunc()` means they can't. In C++, subclasses can still declare a method with the same (simple) name, though.

Let's handle invocations of `sfunc` first. When compiling calls to it, the compiler sees that it isn't declared `virtual`. Thus, it uses the declared type of the object it's being invoked on to determine how to resolve the name. In the first call to `sfunc()` in `main`, the object is a `foo`. So, the compiler starts looking in class `foo` for an `sfunc()`. It finds one, and so compiles basically this:

```
_mangledNameFor_foo::sfunc(pF)
```

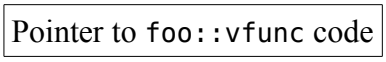
In the second call, it does the same thing. This time it starts at class `bar`, and finds an appropriate sfunc, so compiles basically:

```
_mangedNameFor_bar::sfunc(pB)
```

For virtual functions, the situation is more complicated. The code to be invoked should be the implementation of the method closest to *the acutal class of the object* it's being invoked on. It isn't enough to know the type of the pointer, we need to know what it points at. Unfortunately, we can't know that statically (at compile time). In the example above, even though `pF` is of type `foo*`, the call to `vfunc()` resolves to `bar::vfunc()`, because `pF` is pointing at a `bar`. (Sure, the compiler could figure that out in this case, but it can't always, for instance if a `foo*` is passed to some method as an argument.) We therefore need a way to dynamically determine which instance of `vfunc()` should be invoked.

The solution to this is the use of *vtable*s. A vtable is an array of pointers to methods – an array of the addresses of the first instructions of some methods. We construct the vtable by walking the class hierarchy from base class to some specific class, concatenating the (newly encountered) virtual functions as we go.

The vtable for class `foo` would look like this:



The vtable for class `bar` would be built by starting at class foo. Initially, it'd look just like the one above. The compiler would then descend to class `bar`. It would find a new definition of vfunc, so would replace the entry in the vtable in slot 0. It would also find a second virtual function, so would add it. The vtable for class `bar` would look like this:

Pointer to `bar::vfunc code`
Pointer to `bar::anotherVfunc` code

Just as with instance variables, this means that once a virtual function is declared in a class, the offset in the vtable for that function is the same in every class that includes it (the class where it's originally declared plus all subclasses). That means the compiler can generate correct invocations of virtual functions at compile time, when it doesn't know the class of the object, because it does know the offset of the function in the vtable.

The final details are these. The struct representing the object state (for all classes) has a pointer to the class's vtable as its first field, so at offset 0. After that follow the instance variables, as described above.

When the compiler sees a call like `pF->vfunc()`, it knows the offset of vfunc() in the vtable for all classes that can be considered an object of class foo, because it's the same for all of them (as can be calculated from the information in the `#include`'d .h files). It then generates something like this pseudo-code:

```
r2 ← pF[0]   # fetch pointer to vtable from the object's struct
r2 ← 0(r2)   # fetch pointer to vfunc implementation from the vtable
call (r2)    # invoke the function
```