
# Runtime Time

Apr 1, 2007 • uliwitness


One of the more obscure parts of creating a new programming language is the so-called *runtime*. The runtime, essentially, is the part of your programming language that needs to be added to a compiled program to make it work. The best-known part of the runtime is probably the runtime library. Many programming languages have a runtime library, and it contains things like mathematical functions, common algorithms, and commands to deal with lists and strings.

While in most languages some of these things may actually be built into the language itself instead of being a separate library, the code has to come from somewhere, and usually consists of a pre-written module of code that just gets added to every program you compile. Especially in higher-level programming languages, the runtime library may take on epic dimensions, and include things like a sockets API for internet communications, or a whole 3D engine. Java is best known for having an extensive collection of useful libraries (and, as some people like to point out, maybe a few on top of that). But another thing that is usually part of the runtime in object-oriented programming languages is much less obvious to most people: The object model.

You see, computing (and actually, all of craft itself) has a long history of *bootstrapping*, i.e. building new things based on old things. The mold for a new chisel may be created by using its predecessor on a piece of rock, or whatever. Thus, most people who create a new object-oriented programming language do so in a procedural language. What's misleading is that often people will say things like: "I can write programs in Objective C that I just wouldn't be able to do in C". That's not quite true. There's nothing keeping you from writing object-oriented code using straight C (In fact, early versions of C++ were simply a program that translated a special form of "C with classes" into straight C). It's just that it's a bit of a pain to do so, not to mention that if the OO features aren't part of the language, you usually end up reinventing what you'd get for free and already optimized from an object-oriented language, and every library would invent their own, making inter-operability a bit of a problem.

That part you'd have to reinvent is the *runtime*. To make things easier, I'll just walk you through how you'd write object-oriented code with C. The first thing you'd have to do is create classes. That's easy: In C++, classes are the same as structs, so surely they'll be similar. So, let's have a struct Person:


```
struct Person
{
    char    mName[20];
    int     mAge;
};
```

Now, we want to subclass Person. What we want is something that is a Person, plus something more. Let's create a superhero, who has a super power:


```
struct Superhero
{
    Person mPerson;
    char    mSuperpower[40];
};
```

Now, as it so happens, everyone who expects a Person will find that one can also use an object of this type, as name and age are exactly in the same position, since the Person is first. That is, the above could be rewritten as:

```
struct Superhero
{
    char    mPerson_mName[20];
```

```
    int     mPerson_mAge;
    char    mSuperpower[40];
};
```

A Superhero will thus be usable everywhere a Person is expected, and we don't even need a typecast, we can just pass its mPerson to functions that expect a person. And if we have a case where we are generally satisfied with a Person, but can do something special if it's a Superhero, we can take advantage of the fact that a pointer to mPerson is the same as a pointer to the object itself, and do:

```
void SaveInnocents( Person* saver, int howMany )
{
    if( strcmp( saver->mName, "Clark Kent" ) == 0 ) // We know he's a superhero!
        DoSaveInnocentsWithSuperpower( (Superhero*) saver, howMany );
    else
        DoSaveInnocentsAsBestYouCan( saver, howMany );
}
```

Handy, huh? But of course you're burning to see how to call a function on an object here. Well, that's easy. Where in C++ you would write

```
clarkKent->SaveInnocent( 15 );
```

In our OO-C, you'd simply rewrite that to:

```
SaveInnocent( clarkKent, 15 );
```

So, we simply insert a pointer to `this` as the first parameter. We can even name that parameter `this` or `self` or `me` or whatever your favorite programming language calls it.

Now, it's a little awkward to have each `SaveInnocents()`-function contain a huge block of if-then-else statements to handle all classes. Imagine doing this for the reign of Supermen... there's Clark Kent, Henry Irons, Superboy, the Cyborg, Eradicator, and then all of the Justice League... Surely there's a way to split this across separate functions for each class, like in real OO languages? Yes, there is. What you do is simply use function pointers to remember the address of the function to call:

```
struct Person
{
    char    mName[20];
    int     mAge;
    void (*mSaveInnocentProc)( int howMany );
};
```

And when you create a new Person, you do:

```
struct Person peteRoss;

strcpy( peteRoss.mName, "Pete Ross" );
peteRoss.mAge = 20;
peteRoss.mSaveInnocentProc = DoSaveInnocentsAsBestYouCan;
```

And when you create a new Superhero you do:

```
struct Superhero barryAllen;

strcpy( barryAllen.mPerson.mName, "Barry Allen" );
barryAllen.mPerson.mAge = 30;
strcpy( barryAllen.mSuperpower, "High Speed" );
barryAllen.mPerson.mSaveInnocentProc = DoSaveInnocentsWithSuperpower;
```

And now, whenever you want to save an innocent, what you do is:

```
peteRoss.mSaveInnocentProc( &peteRoss, 15 );
barryAllen.mPerson.mSaveInnocentProc( &barryAllen, 15 );
```

And whaddaya know, except for having to explicitly provide a "this" object, this already looks a lot like C++. But this technique has one huge drawback: Every object carries around a pointer to all its member functions (or "methods"), and there's no way to call through to the superclass easily. What to do? In general, all objects of the same class use the same methods, so the easiest route to go would probably be to just share the list of functions a class understands across all instances:

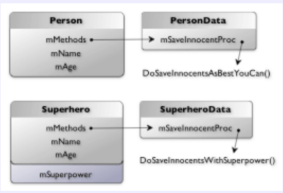
```
struct PersonClassData
{
    void (*mSaveInnocentProc)( int howMany );
};

struct PersonClassData gSharedPersonClassData = { DoSaveInnocentsAsBestYouCan };

struct SuperheroClassData
{
    PersonClassData mPersonMethods;
    // Any methods Superhero adds would follow here.
};
```

```
struct SuperheroClassData gSharedSuperheroClassData = { { DoSaveInnocentsWithSupe
```

Which uses the same "superclass stuff at identical position at start"-approach as we used for the actual objects of our class. And once we have that, we change Person to:




```
struct Person
{
    struct PersonClassData* mMethods;
    char    mName[20];
    int     mAge;
};
```

And now, when we create a new Person, we set its mMethods to point to gSharedPersonClassData, and when we create a new Superhero, we set it to gSharedSuperheroClassData instead. And when we call those two methods, we do:

```
peteRoss.mMethods->mSaveInnocentProc( &peteRoss, 15 );
barryAllen.mPerson.mMethods->mSaveInnocentProc( &barryAllen, 15 );
```

Not too shabby, isn't it? And on top of that, we can now find out what class an object belongs to, by simply comparing mMethods to gSharedSuperheroClassData. But we still can't call through to the superclass. Well, we could just directly look up `mSaveInnocentProc` in gSharedPersonClassData (which is effectively what C++ does these days). But you can also add a field named `super` to the base class's shared class data, which points to NULL if this object really is of the base class, or to the immediate superclass if it's a subclass. Then you can follow each of these pointers up to the base class, if you want to.

There are many more fun things you can do. For example, many languages have special "constructor" and "destructor" functions in the class data that are automatically called when the memory for an object has been allocated, so you can initialize the objects' values, and share the initialization code for inherited variables by calling through to the superclass's function but passing it your class's mPerson or whatever is appropriate.

Also, the fixed layout above is a little problematic because, if you want to add a function to a class that has subclasses, all these subclasses will need to be recompiled because the functions they may have added will have been moved down. While this isn't that big a deal if this is all your program, if you have some libraries that subclass the changed class, you might not be able to recompile them. One way to solve this is to keep the list of functions in a hash map or other kind of associative array, and look them up by name each time you call them, because no matter at what offset in the map your function ends up, the name will stay the same.

Similarly, you can make it possible to add instance variables to a class without having to recompile, or you can keep additional information in your class data, like name of the class, names of functions (so you can save a function name to a file, e.g. a script) and later look up the correct function to call, etc. etc.

If you're curious about this, you may want to check out my Dan's Runtime sample code, which demonstrates how to implement objects in plain C (well, mostly plain C, with a few C++-isms for convenience), as well as how to implement a garbage collector so you won't have to manually allocate and release your objects, and is heavily commented and documented so you can get a second, slightly different explanation of all this.