Dynamic ivars: solving a fragile base class problem

In the "modern" Objective-C runtime (that's iPhone OS or 64-bit Mac OS X), you can dynamically add ivars (instance variables) to a class without declaring them first. This allows a solution to the common "fragile base class" problem involving ivar layouts. Dynamic ivars can also help data hiding and abstraction and can even create a confusing situation where a base class and a sub class have ivars with the same names that don't refer to the same underlying data.

Introduction

The post is about dynamic ivars in Objective-C. Dynamic ivars exist to solve a common type of fragile base class, specifically, the problem of ivar layouts.

A fragile base class problem is a situation where a minor change to the base class can break subclasses. The ivar layout problem is a fragile base class problem where you can't add an ivar to a base class without requiring all subclasses to be recompiled.

The fragile base class problem with ivar layouts occurs because accessing an ivar requires adding the ivar's offset within the object to the object's pointer. Since a subclass' ivars are always positioned after the base class' ivars, the offset to a subclass' ivars must always be the total size of the base class' ivar region plus their own relative offset. Since these offsets are traditionally fixed and unchangeable after compile-time, this means that you can't change the total size of a base class' ivar area without needing to recompile all subclasses to update their ivar offsets.

The reality is that in most object-oriented languages (including Objective-C before dynamic ivars), adding ivars to established classes while maintaining backwards binary compatibility is practically impossible.

Objective-C in conjunction with the Objective-C modern runtime is one of the few compiled language environments to address this problem.

> ***Dynamic*** **does not mean "at any time"**: by dynamic, I mean that the absolute ivar layout is not known at compile-time. Really, I'm talking about ivars that will appear dynamically from the perspective of subclasses (to the base class, they will appear like a normal ivar). While additional ivars can be added to a class at runtime, they can only be added before the class pair is registered (i.e. before there are any instances of the class). See the Apple documentation for class_addIvar for more.

The fragile base class problem of ivar layouts

Let's have a look at how the fragile base class problem of ivar layouts can cause problems by looking at an example.

Adding an ivar to a base class would break all subclasses

Consider an example where a writer of dynamic libraries (e.g. Apple writing the Mac OS X or iPhone OS Cocoa libraries) issues a version of a class that is designed to allow subclasses:

```
@interface LibraryBaseObject : NSObject
{
    NSString *baseObjectIVar;
}
@end
```

Users of that library class then make their own subclasses of the base class:

```
@interface UserSubObject : LibraryBaseObject
{
    NSString * userSubObjectIVar;
}
@end
```

This works fine until the writer of the library wants to add a feature to LibraryBaseObject that requires an additional ivar:

```
@interface LibraryBaseObject : NSObject
{
    NSString *baseObjectIVar;
    id newFeatureObject;
}
@end
```

Traditionally, this would break every existing subclass of `LibraryBaseObject` because no existing subclass would allocate enough memory to hold this new ivar and the offsets to `userSubObjectIVar` would be wrong because these offsets were compiled to include the old instance memory size of `NSObject` + `LibraryBaseObject` and will now offset by the wrong amount.

Yes, code can simply be recompiled with the new headers and all offsets would be corrected to the new values automatically, but until such a recompile, all existing programs that subclass the `LibraryBaseObject` will break.

> Greg Parker's *Hamster Emporium: [objc explain]* has a good post titled Non-fragile ivars with more diagrams showing the ivar layout problem.

Previous workarounds for the fragile base class problem

The common workaround to the fragile base class problem of ivar layouts is to declare your class like this:

```
@interface LibraryBaseObject : NSObject
{
    id private;
}
@end
```

and then actually store all your data in the private class, which you can change in size without affecting the actual `LibraryBaseObject` since the private ivar will only ever be the size of a pointer.

However, this has three problems:

- You must have had the foresight to include this `private` pointer from the beginning
- It involves two dereferences (see the performance note below where I explain that the dereference is the slowest part of ivar access)
- In this untyped scenario, it is unwieldy since all your code must cast this `private` ivar to its actual class before use. You can forward declare a `@class` and use that instead to eliminate this difficulty.

A fix requires that one of the compile-time values becomes dynamic

Previously, I said that we access an ivar by:

1. Add the ivar's offset within the object to the object's pointer value
2. Dereference (read or write from) the memory location referred to by the offset pointer value

This should not be new information to any programmer since Algol: it is the approach taken for accessing data in any `struct`, record or instance variable in most compiled languages.

The problem is that the offset of any subclass' ivars must include the total size of the base class ivar area and none of this can change at runtime.

Obviously, to fix the problem, something must be changeable at runtime. Objective-C's modern runtime fixes this by making the "size of the base class ivar area" a value that can be looked up at runtime.

The "modern" Objective-C runtime therefore requires that accessing an ivar follows this modified approach:

1. Add the offset for the subclass' instance value area to the object's pointer value
2. Add the offset from the subclass' instance area to the ivar
3. Dereference (read or write from) the memory location referred to by the offset pointer value

Once this is done, the base class' ivar area can grow and the subclass' offsets will shift to accommodate this.

> **All ivars are dynamic in the modern runtime**: Since this procedure is followed for all ivars, that means that all ivars in the modern Objective-C runtime are dynamic in that their absolute offsets are never known at compile-time.

Performance note

You may be concerned that the "modern" runtime makes a very common task (accessing ivars) slower.

Yes, adding the extra offset might slow down some code but for most, the reality is that the difference will be unmeasurably small.

Theoretically, if the extra offset needs to be fetched from the struct that describes the class' ivar layout, then the new approach could double time taken. However, the offset for the current subclass will normally already be stored in the `rip` register, which is designed for this type of doubly offset dereference and may reduce the impact of the extra offset to zero. Even if the extra cycle to add the offset is needed, it will be a maximum 20% speed impact, assuming the ivar is in the L1 cache (1 cyle impact versus 4 cycles just to fetch from L1 on newer Intel CPUs), less than 10% for L2 (where the fetch takes 10 cycles) and about 1% for main memory fetches.

In addition to this, compilers already eliminate unnecessary dereferencing, so multiple accesses to the same ivar will be optimized so that this additional offset only occurs once.

Synthesizing ivars

The next question is how do we take advantage of this to add additional ivars to an existing, established class? The reality is that since all ivars are dynamic in the modern runtime, you can simply add extra ivars to a base class and it won't cause a problem for existing subclasses.

But dynamic ivars also enable another way of creating ivars that involves even less disruption to existing classes since it doesn't require changing the header file at all — you can use a synthesized property without a matching ivar. This will create an additional ivar in the implementation that doesn't appear in the declaration that other classes see.

For example, if you start with the following class:

```
@interface MyIvarlessObject : NSObject
{
}
@end
```

You could add dynamic ivars by altering the declaration to:

```
@interface MyIvarlessObject : NSObject
{
}
@property (nonatomic, copy) NSString *myProperty;
@property (nonatomic, copy) NSString *anotherProperty;
@end
```

and add the following to the implementation:

```
@synthesize myProperty=myIvar; // a dynamic ivar named myIvar will be generated
@synthesize anotherProperty;   // a dynamic ivar named anotherProperty will be generated
```

With no matching ivars for either `myIvar` or `anotherProperty`, the `@synthesize` statement will generate a dynamic ivar for each and the `@synthesize` statement becomes the declaration of these ivars.

Since the `@synthesize` statement acts as a declaration, you can now reference `myIvar` or `anotherProperty` in the implementation as though they were regular declared ivars. For example, you could write:

```
- (id)init
{
    self = [super init];
    if (self)
    {
        myIvar = [[NSString alloc] initWithString:@"someString"];
        anotherProperty = [[NSString alloc] initWithString:@"someOtherString"];
    }
    return self;
}
```

If you wanted to hide the property declaration or avoid changing the header file at all, you can declare the properties in a private category in your implementation file instead of in your interface fi le:

```
@interface MyIvarlessObject ()
@property (nonatomic, copy) NSString *myProperty;
@property (nonatomic, copy) NSString *anotherProperty;
@end
```

This will need to appear above the implementation block for `MyIvarlessObject`.

Multiple ivars with the same name

Imagine the following base class:

```
@interface BaseObject : NSObject
{
}
@property (nonatomic, copy) NSString *propertyOne;
@end

@implementation BaseObject
@synthesize propertyOne=myIvar;
@end
```

and a subclass:

```
@interface SubObject : BaseObject
{
}
@property (nonatomic, copy) NSString *propertyTwo;
@end
```

```
@implementation SubObject
@synthesize propertyTwo=myIvar;
@end
```

This base class and the sub class both `@synthesize` an ivar with the name `myIvar`.

In a somewhat unusual quirk, the `myIvar` in `BaseObject` and the `myIvar` in `SubObject` are not the same value — they are actually different ivars, with different offsets within the instance memory area.

This quirk is required to properly address another fragile base class problem: if a `@synthesize` ivar could conflict with a subclass' ivar, then the base class' ivars would still be fragile. To support this idea, `@synthesize` ivars are always `@private`.

However, this does not mean that they are limited to the scope of the `@implementation`. You can access them in category implementations, provided that the category is below the `@implementation` that synthesizes the ivar and hence can see the implicit declaration.

It also means that if the `BaseObject` and `SubObject` were implemented in the same file, they would not compile. Why? Because if they were implemented in the same file, `SubObject` would see the implicit declaration of `myIvar` from its superclass `BaseObject`, attempt to use that instead of synthesizing its own and fail because the `myIvar` from `BaseObject` is `@private`.

Conclusion

You do not normally need to worry about ivar layout problems when adding ivars to your classes. It is only a concern when you are writing or updating a dynamic library and you want to maintain backwards binary compatibility.

Dynamic ivars are a feature of the "modern" runtime; if you are targetting 32-bit Mac OS X, they are not supported.

For other Cocoa platforms, there are numerous reasons why you may want to `@synthesize` ivars:

- They are convenient (don't require an ivar declaration, just a property declaration).
- They allow for greater information hiding from subclasses than `@private` declarations since they don't publicly declare anything.
- You can add them to base classes without needing to recompile subclasses.

Dynamic ivars do require the cost of an extra pointer offset at runtime but you are paying this cost whether you use them or not (it is a required part of the iPhone OS and 64-bit Mac OS X Objective-C runtimes). In any case, you're unlikely to ever notice this cost.

Share this post:

Posted by Matt Gallagher
Monday, March 22, 2010
Filed in categories: Objective-C

Chris L
The iPhone Simulator is a 32-bit Mac environment, which is a shame as the dynamic ivars would save a bit of typing & generally make my code easier to read.
03/22/2010, 06:59:54 - Like - Reply

Matt Gallagher
Yeah, that part is frustrating: you can use dynamic ivars on the iPhone device but you need to wrap everything in #if !TARGET_IPHONE_SIMULATOR because it will fail to compile when targeting the simulator.
03/22/2010, 07:08:58 - Like - Reply

Markus
Hello, nice Article! Thanks!
However,

```
@interface UserSubObject : NSObject
```

should probably be

```
@interface UserSubObject : LibraryBaseObject
```

03/22/2010, 07:16:04 - Like - Reply

Matt Gallagher
Fixed. Thanks for pointing that out.
03/22/2010, 07:22:10 - Like - Reply

Remy "Psy" Demarest
Excuse-me, but there are a few wrong things in this post !
1. @synthesize ivars has nothing dynamic at all, it's completely compile-time, ivars are added to the class at compile-time because @synthesize will also generate the accessors that go along with them, there isn't a special constructor that adds the ivars dynamically at runtime, especially when