How knowing C and C++ can help you write better iPhone apps, part 1

Adrian Kosmaczewski

2010-10-11

While learning how to write iOS applications, you will often encounter the phrase "learn C first"[1]. Writers of Cocoa applications apparently benefit from knowing about C (sometimes even C++), but it is not very clear to many new developers how this actually works.

The obvious question being "why should I learn C if actually I have to use Objective-C to create my apps?", and the answer "because Objective-C is a superset of C" is not really useful, albeit technically correct.

This series of two articles will provide some pointers (no pun intended) to answer that question, in the hopes that new iOS developers will get a copy of the K&R[2] and the Stroustrup[3] books soon.

Objective-C

Before starting with this article, I would like to remind my readers about some key facts of Objective-C, often overlooked by tutorials and teachers, and which explains much of the characteristics of Cocoa as well:

1. All methods (at instance and / or class level) are public, virtual and overridable. You can have @public, @private and @protected instance variables (also known as "ivars"), but methods are public. And yes, you can override class methods in subclasses, too, which is not a very common pattern (you can do that in C++ through templates, though). Polymorphism is the key to understanding Objective-C.[4]
2. There is no formal concept of abstract classes. You can override the alloc and init methods of some class to avoid creating instances from it, but a priori, you can create an instance from almost any class.
3. You can have as many "root classes" as you want. You are not forced to inherit from NSObject, although in most cases that is what you do.

[1]http://stackoverflow.com/questions/180549/learn-c-first-before-learning-objective-c/180832#180832

[2]http://www.amazon.com/Programming-Language-2nd-Brian-Kernighan/dp/0131103628%3FSubscriptionId%3D0F0YTN83N46JSX6KDT02%26tag%3Dakosmasoftwar-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0131103628

[3]http://www.amazon.com/C-Programming-Language-Special/dp/0201700735%3FSubscriptionId%3D0F0YTN83N46JSX6KDT02%26tag%3Dakosmasoftwar-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0201700735

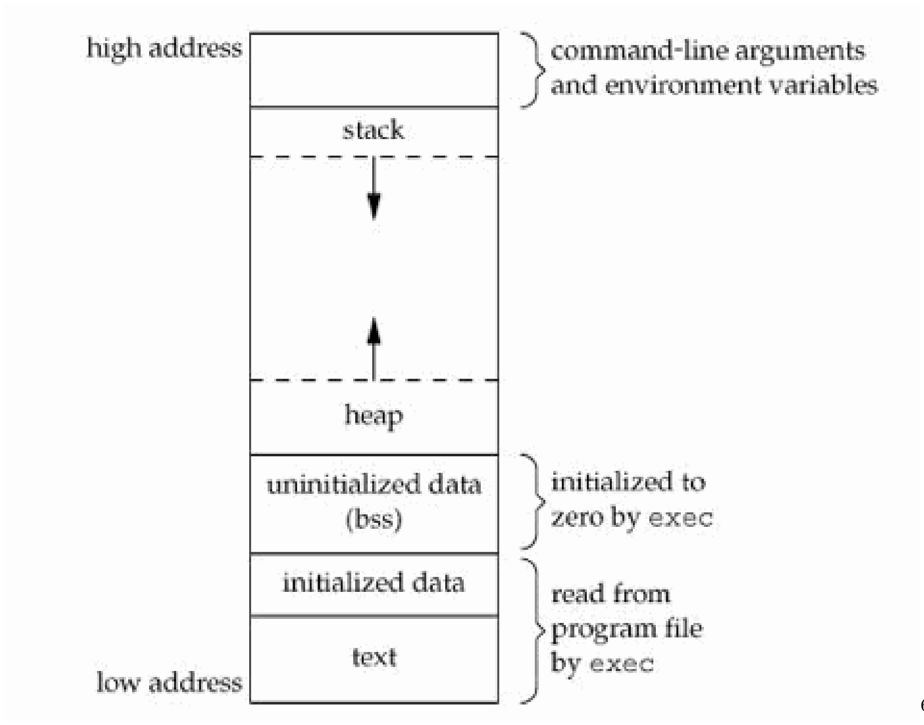
[4]http://en.wikipedia.org/wiki/Type_polymorphism

In Cocoa there are two root classes already defined for developers to use: NSObject and NSProxy.

4. There is no concept of namespaces. That explains the prefix crazyness all over the place, because all classes live in the same planet, so you should better use a nice prefix for your stuff. In my case I usually publish my classes with the "AKO" prefix.

Stack and Heap

This is the probably the most important thing to know about iOS programming. Many developers, particularly those coming from Java, .NET, PHP or other garbage-collected environments, just don't know that the computer memory used by applications is not a uniform space, and that running code uses space in three different, yet complementary, regions of memory, or as computer scientists refer to them, "segments": the text, the stack and the heap.[5]


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

[6]

The "text segment" is where the application code lives in memory while your application runs. Every instruction is there, every single function and procedure and method and executable code lives in that part of memory until your application exits. Usually you don't really have to know anything about the text segment, other that it is there.

When the application starts, the main() function is called and some space is allocated in the "stack". This is another segment of memory allocated for your application, and that's where the memory required for the variables of your

[5]http://www.ualberta.ca/CNS/RESEARCH/LinuxClusters/mem.html

[6]http://www.boundscheck.com/knowledge-base/c-cpp/memory-layout-in-c/342/

functions are allocated. Each time you call a function in your program, a section of the stack called a “frame” is allocated in the stack. The local variables of this new function are allocated there.

[7]

As the name implies, the stack is a “last in & first out” (or LIFO) structure, and when functions call other functions, stack frames are created; when those functions exit, their frames are destroyed automatically (that’s why stack objects are usually referred to as “automatic” objects). In your code, an “end” statement or a closing curly brackets “}” visually indicate a delimiter of stack context; when the current execution goes beyond that element, you can be sure that the stack frame is gone.

The stack region is at the origin of the “stack overflow” concept, which frequently happens when using recursive functions; you allocate too many frames, and you run out of space in the stack. Boom. Another interesting problem associated with stacks is that of “buffer overflows”, which constitutes one of the most common security problems caused by badly written C code. When you write memory contents in the stack, and you don’t check the size of your structures, you might end up writing something on the text segment… which basically means that you might as well be rewriting the code of the application, as it runs. Bad, very bad things can happen then.

[8]

Finally, the “heap” (also called the “data” segment) provides a storage medium that can last throughout the execution of functions; global and static variables live on the heap, until the application exits. To access whatever data you have created in the heap, you require at least one “pointer” on the stack, because that’s how your CPU can access data in the heap; through the stack. You can think of a pointer as just an integer variable that holds the number of a

[7] http://www.futuregov.asia/photologue/photo/2008/aug/30/stack-papers/

[8] http://linguiniontheceiling.blogspot.com/2008/10/thats-madame-trash-heap-to-you.html

particular memory address in the heap (it is actually a bit more complicated, but that's the basic picture).

To summarize: the CPU uses the value of the pointer in the stack to access or, as they say, "dereference", the structure in the heap. Without pointer on the stack, no access to the object on the heap. And here lies the reason of all memory leaks: if you lose your pointer for whatever reason, then you have lost track of an object in the heap. And given that the heap is cleared only when your application exits, you have "leaked" memory, and you will never be able to recover it again (unless you restart your program, that is, but we don't want to ask your user to quit and restart to solve that problem, do we?).

In C++ you can create objects in both the stack and the heap, and to know how to differentiate between both regions, you have to pay attention to the syntax used to create them:

```
// The variable does not use a "star"!
// Created on the stack, don't use "new"
SomeClass stackObject;

// do not call delete stackObject!!!

// Pay attention to the "star"...
// it's a pointer on the stack!
// We are initializing it to NULL,
// as a rule of thumb.
SomeClass *heapObject = NULL;

// And to create the object, use "new"
// to have it created on the heap
heapObject = new SomeClass;

// you must call this to avoid a leak!
delete heapObject;
```

In the case of the stack object, its destructor is called automatically when the stack frame is removed. In the case of the heap object, what goes away is the pointer variable in the stack… but not the object on the heap! To avoid that problem, in C you have to balance every malloc() (or calloc() or similar function) with a free(). In C++ you have to balance every call to "new" with a call to `delete`.

Now, imagine for a second that you have an object in the stack (an automatic object) that points to some other object in the heap, and that our first object is prepared to call the destructor of the second one. This is how the auto_ptr[9] (or "smart pointer") class of the standard C++ template library (STL) works; it takes advantage of the fact that variables on the stack are automatic. Instead of having to manage manually the lifecycle of your objects, an `auto_ptr` object on the stack will call delete automatically when it goes out of scope:

```
#include <iostream>
```

[9]http://en.wikipedia.org/wiki/Auto_ptr

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

int main(int argc, char **argv)
{
    // created on the heap
    SomeClass *heapObject = new SomeClass;

    // now the stackPointer owns the heapObject
    auto_ptr<SomeClass> autoObj(heapObject);

    // Print non-NULL address of heapObject
    cout << *autoObj << endl;
}
```

How does all of this apply to Objective-C? Well, it turns out that the current versions of Objective-C only allow us to create objects on the heap. Apparently creating objects on the stack was possible in early versions of Objective-C, but that isn't the case anymore. All objects that you manipulate in Objective-C live in the heap[10] and as such, if you lose the pointer to them, you have a memory leak. Here goes some code in Objective-C with an obvious memory leak:

```
- (void)doSomething
{
    NSObject *obj = [[NSObject alloc] init];

    // do something else with obj...

    // and now we forget to release, and boom:
    // Memory leak! This is because the
    // "obj" pointer in the stack is lost
    // when this stack frame is removed...
    // ... but the heap object stays!
}
```

What can you do to avoid that? Two things:

1. Remember how in C you balance malloc() with free, and how you balance "new" with "delete" in C++? Well, in Objective-C you have to balance every call to "alloc", "copy" or "retain" with a "release".
2. You can also use the "autorelease" method, to add your object to the current autorelease pool, so that it will be automatically disposed of in the next iteration of the run loop cycle. OK, this is slightly more complex, but basically this also solves the memory leak problem.

Here goes some code illustrating the two solutions in detail:

```
// First solution
- (void)doSomething
```

[10]Well, that is not entirely true; there is one kind of Objective-C object that lives in the stack, and that is blocks[11]. But this is an exception, one that I will tackle in a separate blog post.

```
{
    NSObject *obj = [[NSObject alloc] init];

    // do something else with obj...

    [obj release];
}

// Second solution
- (void)doSomething
{
    NSObject *obj = nil;
    obj = [[[NSObject alloc] init] autorelease];

    // do something else with obj...

    // Don't release obj! The object referenced
    // by that pointer will be disposed
    // in the next run loop iteration.
}
```

However, please, please, avoid the following code, which stretches the use of the autorelease pool to the level of abuse[12]:

```
for (NSData *data in veryLargeArrayWithImages)
{
    UIImage *image = [UIImage imagedWithData:data];
    // do something with image
}
```

because this will create an unnecessary large number of instances that will remain in memory until the next run loop cycle (which will probably happen after the end of the current loop, that is, in a long CPU time in the future). Instead, either use a temporary autorelease pool, or use the non-autoreleased version of the code above (which I prefer, particularly in the iPhone):

```
for (NSData *data in veryLargeArrayWithImages)
{
    UIImage *image = [[UIImage alloc] initWithData:data];
    // do something with image
    [image release];
}
```

All of these memory management techniques are explained in detail in my article 10 iPhone Memory Management Tips[13].

Mike Ash has written an extensive article about stack and heap objects[14] that will help you understand all of these issues even better. Finally, to see visually how the stack and the heap collaborate with each other, and how to keep

[12]Autorelease pools are definitely material for another blog post.
[13]/blog/10-iphone-memory-management-tips/
[14]http://www.mikeash.com/pyblog/friday-qa-2010-01-15-stack-and-heap-objects-in-objective-c.html

an object-oriented mindset when writing code in C, I recommend watching on iTunesU the excellent Programming Paradigm[15] course by professor Jerry Cain from Stanford University.

Object ownership

Ownership is a key concept in C++ memory management: whenever an object creates or copies another, the first one becomes the "owner" of the second. And as such, it is the task of the first one to destroy the second. Here goes some code that shows that pattern; here some class requires another one, and the ownership relationship is enforced throughout the lifetime of both objects.

```
// Forward declaration of some owned class
class Owned;

// Declaration of the owner
class OwnerClass : BaseClass
{
public:
    OwnerClass();
    virtual ~OwnerClass();
    const Owned& getOwnedObject();

private:
    Owned* _owned;
};

// Constructor implementation
OwnerClass::OwnerClass()
: BaseClass()
, _owned(new Owned)
{
}

// Destructor implementation
OwnerClass::~OwnerClass()
{
    delete _owned;
}

// Provide to clients a reference
// to the underlying owned object
const Owned& OwnerClass::getOwnedObject()
{
    return _owned;
}
```

In Objective-C, the chain of ownership is important to solve the problem of circular references; this is a common problem when creating delegation relation-

[15]http://itunes.apple.com/WebObjects/MZStore.woa/wa/viewPodcast?id=384233005

ships between objects. When an object is delegate of another, the reference between the object and its delegate must be weak:

```
// A delegate protocol declaration
@protocol OwnedDelegate <nsobject>

@optional
- (void)ownedObjectSaysSomething:(Owned *)owned;

@end

// Interface of the owner
@interface OwnerClass : NSObject <OwnedDelegate>
{
@private
    Owned *_owned;
}

@property (nonatomic, retain) Owned *owned;

@end

// Interface of the owned
@interface Owned : NSObject
{
@private
    id<OwnedDelegate> _delegate;
}

@property (nonatomic, assign) id<OwnedDelegate> delegate;

@end
```

Pay attention at the @property declarations in the code above. The OwnerClass "retains" the Owned object, but the Owned class just "assigns" the delegate. This way, we know clearly who's responsible of releasing who: it is the OwnerClass dealloc method that will do that, and not the other way around!

```
@implementation OwnerClass

@synthesize owned = _owned;

- (id)init
{
    if (self = [super init])
    {
        _owned = [[Owned alloc] init];
        _owned.delegate = self;
    }
    return self;
}
```

```
- (void)dealloc
{
    _owned.delegate = nil;
    [_owned release];
    _owned = nil;

    [super dealloc];
}

#pragma mark -
#pragma mark OwnedDelegate protocol

- (void)ownedObjectSaysSomething:(Owned *)owned
{
    // do something now
}

@end
```

If you don't follow this pattern, you might end up with a circular reference, and the reference counting scheme of Objective-C won't be able to solve this problem, which means that you might end up leaking two instances at once!

free() vs. delete vs. dealloc

In C, free() is a function taking any pointer as parameter, and whose purpose is to mark the region of memory pointed by the parameter as available for the application. It is up to the application developer to pay attention to properly clean any owned objects in the structures stored in that region, as the standard C runtime makes no assumptions whatsoever about those secondary relations.

In C++, delete is an operator, and its task is to ultimately call the destructor of the target object, starting by the class of the current object, until it reaches the last class in the inheritance chain, and it does this immediately when called, in a synchronous fashion. Your constructors do not have to call the constructor of the base class in their implementation; it is, however, strongly recommended to mark your destructors as virtual[16], to ensure that the delete operator does this in all cases.

On the other hand, dealloc in Objective-C is a polymorphic, virtual, overridable method, not an operator neither a function, and it is never called directly by your code. It is called in a kind of asynchronous fashion, when the current run loop ends, and when the retain count of the current object reaches zero. This means that you might not know exactly when that happens, and actually, you shouldn't care. The important thing is to release all the things you were holding on to (see the "object ownership" section above).

Another important fact of dealloc is that, being polymorphic, only the implementation of the current class will be called, so you must always remember to

[16]http://www.parashift.com/c++-faq-lite/virtual-functions.html#faq-20.7

add a call to [super dealloc] at the end of your own dealloc. And always, always add it at the end, not at the beginning[17], of your own dealloc.

Conclusion

I hope this information will be useful to developers new to the iOS platform. Objective-C and Cocoa share many characteristics with other similar platforms and frameworks written in C and C++, and developers working with garbage-collected runtimes are not used to know the layout of structures in memory. I think, however, that knowing this is useful also in such environments.

In the second part of this series I will dive into variable initialization, bitwise masks, functions and other related aspects of C and C++ programming, and how they relate to Objective-C.

Acknowledgement: Many thanks to Ariel Rodríguez[18] for reviewing early drafts of this post.

Update, 2010-10-13: Ariel has written an excellent follow up to this article, explaining the problem of stack overflows[19].

[17]http://stackoverflow.com/questions/909856/why-do-i-have-to-call-super-dealloc-last-and-not-first/909925#909925
[18]http://volonbolon.net/
[19]http://volonbolon.net/post/1305559150/stack-overflow