# Root Class

Unlike C++, Objective-C deﬁnes a root class. Every new class should be a descendant of the root class. In Cocoa, that class is NSObject which provides a huge number of facilities for the run-time system.

Therefore, every object can be of type NSObject, and **every pointer to an object can be declared as NSObject\***.

# New Data Types

## BOOL

// C++

bool isSuccess = true; // opposite: false

// Objective-C

BOOL isSuccess = YES; // opposite: NO

## nil and Nil

// In C++

int\* p = NULL;

Foo\* foo = NULL; // Foo is a class

// In Objective-C

int\* p = nil; // pointer to an object

Foo\* foo = Nil; // pointer to a class

Note: nil and NULL should not be interchangeable.

Read more: [Nullability](https://developer.apple.com/swift/blog/?id=25) in Objective-C and [here](https://stackoverflow.com/a/33682230).

## id

idis a generic type for pointers. You should use id(Objective-C pointer) instead of void\* (raw pointer)because idis managed by **ARC**.

## SEL

SELis a generic type for selectors (like **function pointers** in C++)

You generally create a selector by using the keyword @selectoralong with the function name and a number of “:” depending on how many parameters can be passed. A selector is actually a string, which binds to a method identifier at runtime.

More details!

SEL s = @selector(foo:); // a pointer to a function foo that takes 1 parameter

# #import vs #include

// In C++

#pragma once

#include "Rectangle.h"

// In Objective-C

#import "Rectangle.h"

#import is identical to #include, except that it makes sure that the same file is never included more (so we don’t need #pragma once, like C/C++)

EX:

|  |  |
| --- | --- |
| **Objective-C** | |
| **In file Foo.h** | **In file Foo.m** |
| @interface Foo : NSObject  {  ...  }  @end | #import "Foo.h"  @implementation Foo  ...  @end |

# Function

## Declaration and Calling

|  |  |
| --- | --- |
| **In C++** | **In Objective-C** |
| **/\* Declaration \*/**  int foo() {}  static void bar(int a, char b)  {} | **/\* Declaration \*/**  // It’s called *instance method* in Objective-C  - (int)foo {}  // And it’s called *class method*  + (void)bar: (int)a: (char)b {} |
| **/\* Calling \*/**  CppObject\* ptr = ...;  ptr->foo(5,3);  CppObject::bar(5,3); | **/\* Calling \*/**  NSObject\* ptr = ...;  [ptr foo:5:3];  [NSObject bar]; |
|  | **/\* labeled arguments \*/**  // Declaration  - (void)insertObject: (id)anObject atIndex:(int)index  // Calling  [shelf insertObject:book atIndex:2]; |

Note: The ﬁrst parameter cannot have a label.

Differences:

* Objective-C functions can have **labeled**arguments which make it more clear which parameter gets what value. In theory, labeled arguments would also allow the programmer to pass parameters in any order; however, Objective-C requires the same order as in the declaration.
* Unlike C++, it's very fine for Objective-C for pointer to be nil, in which case the "call" is ignored (where, in C++, it would throw a pointer violation exception). This makes it possible to eliminate checks against nil objects.

Similarity:

Objective-C allows the dot (.) operator that offers an alternative to square bracket notation ([]) to invoke methods, just like C/C++. For example:

// You can:

myInstance.setValue(10);

printf("myInstance value: %d", myInstance.getValue());

// Instead of:

[myInstance setValue:10];

printf("myInstance value: %d", [myInstance getValue]);

**self and super**

Objective-C provides two hidden parameter that can be used within a method definition to refer to the object that performs the method:

* self is the current object (like this in C++)
* super is the parent class

EX:

@interface Foo : NSObject

{

int x; // It’s called instance variable in Objective-C

int y;

}

- (void)f;

@end

@implementation Foo

- (void)f

{

x = 1;

self->y = 3; // both ways are valid

}

@end

## Overloading

In C++, two functions are overload when one has different parameter types from the other.

In Objective-C, overloaded functions are only allowed when they have different **parameter labels**.

|  |  |
| --- | --- |
| **In C** | **In Objective-C** |
| class Foo  {  public:  int g(int); // OK  int g(float); // OK  int g(float) const; // OK  }; | @interface Foo : NSObject  {  }  - (int)g:(int)x;  - (int)g:(float)x; // Error  - (int)g:(int)x :(int)y; // OK  - (int)g:(int)x :(float)y; // Error  - (int)g:(int)x andY:(float)y;//OK  @end |

## Function Pointer

Refer to 3.3.5

Objective-C member functions are "messages" and when, in Objective-C we say that the **receiver**(the pointer) **responds to a selector,**it means that it implements the virtual function we are trying to call. When there is an interface, C++ objects must implement all of its member functions. In Objective-C this is not required, so we can send a "message" to somewhere that does not necessarily implement it (so an exception is raised).

// Objective-C

NSObject\* ptr = ...; // some pointer

if ([ptr respondsToSelector:@selector(foo::)] { // Check if foo is available

[ptr foo:5:3];

}

Now we are sure that the receiver responds to the selector, so we can call it. In C++ this check is not needed, because implementations must always "respond to the selector", or the source does not compile. Note that we must know how many parameters the selector gets; that’s why we used 2 “:” in the @selector because foo takes 2 arguments.

# Properties

One of the primary principles in OOP is that an object should hide its internal workings behind its public interface. In other words, it’s important to access an object’s properties using behavior exposed by the object rather than trying to gain access to the internal values directly.

Objective-C properties offer a way to define the information that a class is intended to encapsulate.

## Properties Declaration

The general form a property declaration is:

@property (attributes) type name;

For example:

@interface Person : NSObject

{

// We can do this, but it’s NOT A GOOD WAY to encapsulate objects in OOP

@public

NSString \*fullName;

}

// We SHOULD do this, instead:

@property NSString \*fullName; // Note: cannot put properties in { }

@end

## Using Accessors to Get or Set Properties’ Values

You can think of a property declaration as being equivalent to declaring two accessor methods:

@property NSString\* fullName;

// This is equivalent to (which generated automaticall by the compiler):

@private NSString\* \_fullName;

- (NSString\*)fullName {

return \_fullName;

}

- (void)setFullName:(NSString\*)inFullName {

if (\_fullName != inFullName) {

\_fullName = inFullName;

}

}

To use properties in Objective-C, we have to follow specific naming conventions:

* The **getter** has the same name as the property.
  + In the case of BOOL properties, it’s customary for the getter method to start with the word "is". For example:

@property (getter=isFinished) BOOL finished;

// This just gives a custom name for a getter.

// The default getter is just the name of the property.

* The **setter** has the name that starts with the word "set" and then uses the capitalized property name:

@property (setter=setFullName:) NSString \*fullName;

// This just gives a custom name to the setter.

// The default setter is just the capitalized property name with "set" as a prefix

// So in this example, we can *skip* (setter=setFullName:)

Example:

**// In main.m:**

#import "Foo.h"

int main(int argc, const char \* argv[])

{

Foo\* p\_Foo = [[Foo alloc] init];

[p\_Foo setFullName:@"Tri Ho"];

NSLog(@"My full name is %@", [p\_Foo m\_FullName]);

return 0;

}

**// In Foo.h:**

@interface Foo : NSObject

@property (weak, setter=setFullName:) NSString\* m\_FullName;

@end

## Property Attributes

### Writability

// Allow getter but not setter

@property (readonly) NSString\* fullName;

// Default. No need to specify this attribute explicitly

@property (readwrite) NSString\* fullName;

// For multiple attributes, simply include them as a comma-separated list

@property (readonly, getter=isFinished) BOOL finished;

### Setter Semantics

* strong (default, under [ARC](#_2xcytpi) only): Specifies that there is a strong reference (owning relationship) to the destination object. When a strong reference is declared, the reference counter increases by one.
* weak (under ARC only): Specifies that there is a weak reference (non-owning relationship) to the destination object. If the destination object is deallocated, the property value is automatically set to nil. Because weak references do not increase the reference counter, objects that are only held with a weak reference(s) are automatically released since the counter will be set to 0.

More details (or you can read the [ballon](https://stackoverflow.com/a/18344946) comparison here)

A strong reference means that you want to "own" the object you are referencing with this property/variable. The compiler will take care that any object that you assign to this property will not be destroyed as long as you point to it with a strong reference. Only once you set the property to nil will the object get destroyed (unless one or more other objects also hold a strong reference to it).

// In .h

@property NSString\* prop;

// In .m

NSString\* localVar = @"abc"; // localVar refers to a memory block holding "abc".

// Because localVar is 'strong', it’s the owner of the block.

// And it’s responsible for releasing the block.

prop = localVar; // prop also refers to this block.

// Because it’s 'strong', it becomes the 2nd owner of the block.

localVar = nil; // Under ARC, the block will be release when this happends.

prop = nil;

A weak reference means that you don't want to have control over the object's lifetime. The object you are referencing weakly only lives on because at least one other object holds a strong reference to it. Once that is no longer the case, the object gets destroyed and your weak property will automatically set to nil.

// In .h

@property (weak) NSString\* prop;

// In .m

NSString\* localVar = @"abc"; // localVar refers to a memory block holding "abc".

// Because localVar is 'strong', it’s the owner of the block.

// And it’s responsible for releasing the block.

prop = localVar; // prop also refers to this block.

// But this time, it’s 'weak' and not 2nd owner of the block.

localVar = nil; // Under ARC, the block will be release when this happends.

The most frequent use cases of weak references are:

1. Delegate properties (which are often referenced weakly to avoid retain cycles - Eg. obj1 retains obj2 and obj2 retains obj1)
2. Outlets
3. Subviews / Controls (of a view controller's main view because those views are already strongly held by the main view).

* copy: Specifies that a copy of the object should be used for assignment. The previous value is sent a release message. The copy is made by invoking the copy method. This attribute is valid only for object types, which must implement the NSCopying protocol.
* assign: Specifies that the setter uses simple assignment. This attribute is the default. You use this attribute for scalar types such as NSInteger and CGRect.
* retain: Specifies that retain should be invoked on the object upon assignment. The previous value is sent a release message.

Tip:

In OS X v10.6 and later, you can use the [\_\_attribute\_\_](https://nshipster.com/__attribute__/) keyword to specify that a Core Foundation property should be treated like an Objective-C object for memory management:

@property \_\_attribute\_\_((NSObject)) CFDictionaryRef myDictionary;

### Atomicity

* atomic (default): Synthesized accessors provide robust access to properties in a multithreaded environment—that is, the value returned from the getter or set via the setter is always fully retrieved or set regardless of what other threads are executing concurrently.
* nonatomic: Specifies that accessors are nonatomic.

If you specify strong, copy, or retain and do not specify nonatomic, then in a reference-counted environment, a synthesized getter for a property uses a lock and retains and autoreleases the returned value. The implementation will be similar to the following:

[\_internal lock]; // lock using an object-level lock

id result = [[value retain] autorelease];

[\_internal unlock];

return result;

If you specify nonatomic, a synthesized accessor for an object property simply returns the value directly.

**@synthesize**

If you use: @synthesize prop = \_prop; the compiler creates:

* A \_prop instance variable, if it doesn't already exists.
* A prop getter method, if you haven't implemented it;
* A setProp setter method, if you haven't implemented it.

If you use @synthesize prop; the compiler creates a prop instance variable if it doesn't already exists, instead of \_prop.

If you don’t use @synthesize and have no implementation for the getter/setter, the getter/setter will be autosynthesized (by Clang compiler), therefore a \_prop instance variable will be created. So basically, **if you use new version of Clang, you do NOT need @synthesize anymore**. Other exceptions when autosynthesized feature is NOT applied:

* Properties with readonly with custom getters
* Properties declared in @protocol
* Properties declared in a category (because categories cannot create ivars)
* Overridden properties (when you override a property of a superclass, you must explicitly synthesize it)

Note: If you explicily implement both getter and setter, so the autosynthesis doesn't synthesize any (as said above), so it doesn't create the instance variable. You have to declare the \_prop yourself.

**@dynamic**

# Memory Management

In C++, variables are “automatic” by default: unless they are declared static, they only exist inside their definition block. Only dynamically allocated memory is usable beyond, until the matching free() method or delete operator is called. Objects follow this rule in C++.

However, in Objective-C, all objects are dynamically allocated. That’s because **all objects in Objective-C are allocated in the Heap (except for blocks which are allocated in the Stack)**. This is rather logical, C++ being very static, and Objective-C being very dynamic. Objective-C’s dynamism wouldn’t be that rich if objects weren’t created at run-time.

One more thing, **references (&) do not exist in Objective-C**. Because the objects are always dynamically created, there are only referenced as pointers.

## Basic Memory Management Rules

The memory management model is based on **object ownership** (or a reference counting scheme). Any object may have one or more owners. As long as an object has at least one owner (or if its retain count is greater than 0), it continues to exist (it’s *alive*). If an object has no owners (or if its retain count drops to 0), it will be destroyed by the system.

To make sure it is clear when you own an object and when you do not, Cocoa sets the following policy:

### You own any object you create

You create an object using a method whose name contains with [“alloc”, “new”](#_1y810tw), “copy”, or “mutableCopy”.

NSString\* obj1 = [[NSString **alloc**] initWithString:@"hello"];

NSString\* obj2 = [someOtherString **copy**];

NSMutableString\* obj3 = [someOtherString **mutableCopy**];

NSString\* obj4 = [NSString **new**];

[obj1 release];

[obj2 release];

[obj3 release];

[obj4 release];

### You own any object using retain

When you call retain on an object, you increase its retain count (RC) by 1.

A received object is normally guaranteed to remain valid within the method it was received in, and that method may also safely return the object to its invoker. You need to retain object in two situations: (1) To prevent an object from being being disposed of. (2) In the implementation of an accessor method or an init method, to take ownership of an object you want to store as a property value.

### When you no longer need it, you must relinquish ownership of the object you own

You relinquish ownership of an object by sending it a release or autorelease message, which decreases the RC by one.

**When the RC drops to 0, the object will deallocate itself** by calling [dealloc](#_4i7ojhp) (under ARC). This means you should never call dealloc directly. Just release the object correctly, and it will handle everything itself.

NSString\* obj1 = [[NSString alloc] init]; // Rule 1

[obj1 release];

[obj2 retain]; // Rule 2

[obj2 release];

// Can own the same object many times

Pidgeon\* pidgeon = [[Pidgeon alloc] init]; // Rule 1

[pidgeon retain]; // Rule 2

// RIGHT

[pidgeon release]; // Rule 3

[pidgeon release];

// WRONG

[pidgeon dealloc]; // Rule 3

[pidgeon dealloc];

### You must not relinquish ownership of the object you don’t own

This is just corollary of the previous policy rules, stated explicitly.

One common mistake is to think "I have to release this string because I created it with [NSString stringWithString:@"hello"]." Look at Rule 1 again. stringWithString: does not start with "alloc", "copy" or "new", which means you don't own the object, so don't release it. Or else, this would cause a crash.

**Remember:**

NSString\* objRef = [[NSString alloc] init];

**An object doesn’t mean a reference:**

Have to understand that objRef is NOT acutally an object. It is a reference to an object of type NSString. Objects have retain counts, but references to objects do not.

**References have the ownership, not objects:**

objRef is the reference to an object of type NSString, and it’s also the owner of that object. So being an 'owner' is the capability of a reference, not an object.

**But only objects have the retain count, not references:**

*Question*: Suppose I have two objects objectARef & objectBRef of the same type. The objectARef refers to object having RC of 1, and the objectBRef refers to object having RC of 0. Now I am performing operation: objectBRef = [objectARef retain];

Am I right that now the RC of objectARef = 2 & objectBRef = 1.

*Answer*: Wrong! The objectARef initially refers to the object having RC of 1. After 'objectBRef = objectARef retain', the RC (of the object that objectARef refers to) increases to 2. Also, both objectBRef and objectARef refer to the same object. Since RCs are owned by the object, there is only one object and only one RC. It is 2.

**Retain count = reference count = number of *strong* references to an instance**

**Become an object’s owner = increase object’s retain count by 1**

**Passing Parameters by Pointer**

When a function receives an object as a parameter, the receiver does NOT initially own the object. There is therefore no need to release it. The object may therefore be deallocated at any time — unless the receiver takes ownership of if (eg, using CFRetain). If this happens, when the receiver has finished with the object — either because it is replaced a new value or because the receiver is itself being deallocated — the receiver is responsible for relinquishing such ownership (eg, using CFRelease).

**Return and Ouput Parameters**

Some methods in Cocoa specify that an object is returned by reference (that is, they take an argument of type ClassName\*\* or id\*). A common example is the NSError object as illustrated by:

NSError\* error;

NSString\* string = [[NSString alloc] initWithContentsOfFile:fileName

encoding:NSUTF8StringEncoding

error:&error];

// use string and error ...

[string release]; // Rule 3

// [error release]; // Rule 4

In these cases, the same rules apply as the first section. When you invoke any of these methods, you do not create the NSError object, so you do not own it. In other words, you do NOT own objects returned by reference. There is therefore no need to release it.

## Constructor

In C++, we use the new operator to dynamically allocate a block of memory for an instance of the receiving class. To initialize this block of memory, we can use new() (if the compiler supports) or memset.

In Objective-C, that +new() is equivalent to +alloc/-init in Apple's NSObject implementation. But because "explicit is better than implicit", the Objective-C community generally avoids +new().

That’s mean Objective-C also support +new(). The differences between these functions are:

* +alloc: Class method of NSObject. It allocates a block of memory to hold the object and returns a pointer to the new instance of the object.
* -init: Instance method of NSObject. It initializes the new object after memory for it has been allocated.
* +new(): Class method of NSObject. It allocates a new instance of the object, sends it an init message, and returns the initialized object.

So, new = alloc + init. And alloc should always be followed by init.

Note: In C++, the name of the constructor cannot be chosen. In Objective-C, the initializer is a method like any other, and has been **defined and called explicitly**, and the ‘init’ prefix is traditional but not mandatory. However, you are strongly encouraged to follow a rule: **the name of an initialization method must begin with ‘init’**.

Example:

|  |  |
| --- | --- |
| **In C++** | **In Objective-C** |
| class Point  {  public:  Point(int x, int y);  private:  int x;  int y;  };  Point::Point(int inX, int inY)  {  x = inX;  y = inY;  }  ...  // must be delete later  Point\* p1 = new Point(5, 6);  ...  delete p1;  // no need to delete  Point p2(3,4); | @interface Point : NSObject  {  int x;  int y;  }  -(id) initWithX:(int)inX andY:(int)inY;  @end  @implementation Point  -(id) initWithX:(int)inX andY:(int)inY  {  // An initializer of the superclass must be called  // If the superclass is NSObject, this must be init  if (!(self = [super init])) {  return nil; // In case of failure, return nil  }  // In case of success, make additional initializations  self->x = inX;  self->y = inY;  return self; // and return the object itself  }  @end  ...  // must be delete later  Point\* p1 = [[Point alloc] initWithX:5 andY:6];  ...  [p1 release]  // no need to delete  Point\* p2 = initWithX:3 andY:4 |

## Destructor

Without [ARC](#_2xcytpi), Objective-C uses retainand releasemethods as its destructors.

With ARC, the destructor is an instance method named dealloc defined in NSObject. It is called automatically to deallocate objects, and it **cannot be called explicitly**, just like in C++, only the way to release the object is different.

The role of dealloc is to free the object's own memory, and to dispose of any resources it holds, including **ownership** of any object instance variables.

|  |  |
| --- | --- |
| **In C++** | **In Objective-C** |
| class Point  {  private:  char\* firstName;  char\* lastName;  public:  ~Point();  };  Point::~Point() {} | @interface Point : NSObject  {  NSString \*firstName;  NSString \*lastName;  }  @end  @implementation Point  -(void) dealloc // here we override dealloc from NSObject, so no need to put its prototype in .h file.  {  [firstName release];  [lastName release];  // don’t forget to transmit to the superclass  [super dealloc];  }  @end |

# ARC (Automatic Reference Counting)

## What Is ARC?

ARC (Automatic Reference Counting) is a **compile time**feature (of the Clang compiler) that tells the compiler "here are my objects: please figure out when they have to be destroyed". With ARC you do NOT need to send retain/release/autoreleasemessages to your objects; the compiler does that automatically.

In particular, ARC automatically inserts the appropriate retain, release and autoreleaserequired for reference counting at compile time, by applying the [memory management rules](#_3j2qqm3) that all Objective-C developers have had to use over the years. The compiler also generates appropriate dealloc methods for you. This frees the developer from having to manage themselves. Because these methods are inserted at compile time, no collector process is needed to continually sweep memory and remove unreferenced objects.

*How do I think about ARC?*

Try to stop thinking about where the retain/release calls are put and think about your application algorithms instead. **Think about strong and weak pointers in your objects, about object ownership, and about possible retain cycles.**

## ARC Enforces New Rules

To work, ARC imposes some new rules that are not present when using other compiler modes. If you violate these rules, you get an immediate compile-time error.

* You cannot explicitly call dealloc, or override or invoke retain, release, or autorelease. The prohibition extends to using @selector(retain), @selector(release), and so on.
* You can override a dealloc if you need to manage resources other than releasing instance variables. In this case, the custom dealloc under ARC doesn’t require a call to [super dealloc] (it actually results in a compiler error). The chaining to super is automated and enforced by the compiler.
* You can still use CFRetain, CFRelease, and other related functions with Core Foundation-style objects (see [Managing Toll-Free Bridging](https://developer.apple.com/library/archive/releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011226-CH1-SW1)).
* You cannot use NSAllocateObject or NSDeallocateObject. The only way to create objects under ARC is using alloc.
* You cannot use [NSAutoreleasePool](about:blank) objects. ARC provides @autoreleasepool (even works in non-ARC code) blocks instead. These have an advantage of being 6x faster than NSAutoreleasePool.
* You cannot use object pointers in C structures. Rather than using a struct, you can create an Objective-C class to manage the data instead.
* There is no casual casting between id and void\*.
* ARC requires you to assign the result of [super init] to self in init methods.

// The following is INVALID in ARC init methods:

[super init];

// The simple fix is to change it to:

self = [super init];

// The proper fix is to do that:

[if (self = [super init]) {...}](#_1y810tw)

* You cannot use memory zones. There is no need to use NSZone any more — they are ignored by the modern Objective-C runtime anyway.
* To allow interoperation with manual retain-release code, ARC imposes a constraint on method naming. That is, you cannot give an accessor a name that begins with 'new'. This in turn means that you can’t, for example, declare a property whose name begins with 'new' unless you specify a different getter:

// Won't work:

@property NSString \*newTitle;

// Works:

@property (getter=theNewTitle) NSString \*newTitle;

* “Assigned” instance variables become strong.

Before ARC, instance variables were non-owning references—directly assigning an object to an instance variable did not extend the lifetime of the object. To make a property strong, you usually implemented or synthesized accessor methods that invoked appropriate memory management methods; in contrast, you may have implemented accessor methods like those shown in the following example to maintain a weak property.

@interface MyClass : Superclass {

id thing; // Weak reference.

}

// ...

@end

@implementation MyClass

- (id)thing {

return thing;

}

- (void)setThing:(id)newThing {

thing = newThing;

}

// ...

@end

With ARC, instance variables are strong references by default—assigning an object to an instance variable directly does extend the lifetime of the object. The migration tool is not able to determine when an instance variable is intended to be weak. To maintain the same behavior as before, you must mark the instance variable as being weak, or use a declared property.

@interface MyClass : Superclass {

id \_\_weak thing;

}

// ...

@end

@implementation MyClass

- (id)thing {

return thing;

}

- (void)setThing:(id)newThing {

thing = newThing;

}

// ...

@end

Or:

@interface MyClass : Superclass

@property (weak) id thing;

// ...

@end

@implementation MyClass

@synthesize thing;

// ...

@end

* You can't use strong id in C structures.

For example, the following code won’t compile:

struct X { id x; float y; };

This is because x defaults to strongly retained and the compiler can’t safely synthesize all the code required to make it work correctly. For example, if you pass a pointer to one of these structures through some code that ends up doing a free, each id would have to be released before the struct is freed. The compiler cannot reliably do this, so strong ids in structures are disallowed completely in ARC mode. There are a few possible solutions:

* 1. Use Objective-C objects instead of structs. This is considered to be best practice anyway.
  2. If using Objective-C objects is sub-optimal, (maybe you want a dense array of these structs) then consider using a void\* instead. This requires the use of the explicit casts, described below.
  3. Mark the object reference as \_\_unsafe\_unretained. This approach may be useful for the semi-common patterns like this:

struct x { NSString \*S; int X; } StaticArray[] = {

@"foo", 42,

@"bar, 97,

...

};

You declare the structure as:

struct x { NSString \* \_\_unsafe\_unretained S; int X; }

This may be problematic and is unsafe if the object could be released out from under the pointer, but it is very useful for things that are known to be around forever like constant string literals.

## When ARC Doesn’t Work?

ARC does not manage the lifetime of objects created by malloc/free (C++ objects, Core Foundation objects, file descriptors, etc.), so you need to manage (retain, release) them manually, or you can transfer their ownership to ARC for automatic memory management.

## ARC Introduces New Lifetime Qualifiers

ARC introduces several new property attributes and lifetime variable qualifiers for objects. You should take advantage of these qualifiers to manage the object graphs in your program.

**Note**: ARC does not guard against *strong reference cycles* (previously known as retain cycles — see [Practical Memory Management](about:blank)). Judicious use of *weak* relationships will help to ensure you don’t create retain cycles.

### Property Attributes

Read [this section](#_44sinio) for more details.

The keywords weak and strong are introduced as new declared property attributes, as shown in the following examples:

// The following declaration is similar to "@property (retain) MyClass \*myObject;"

// Under ARC, strong is the default for object types

@property (strong) MyClass \*myObject;

// The following declaration is similar to "@property (assign) MyClass \*myObject;"

// except that if the MyClass instance is deallocated,

// the property value is set to nil when there are no strong references to the object,

// instead of remaining as a dangling pointer.

@property (weak) MyClass \*myObject;

### Variable Qualifiers

They are:

* \_\_strong is the default. An object remains “alive” as long as there is a strong pointer to it.
* \_\_weak specifies a reference that does not keep the referenced object alive. A weak reference is set to nil when there are no strong references to the object.
* \_\_unsafe\_unretained specifies a reference that does not keep the referenced object alive (like \_\_weak) and is *not* set to nil when there are no strong references to the object. If the object it references is deallocated, the pointer is left dangling.
* \_\_autoreleasing is used to denote arguments that are passed by reference (id\*) and are autoreleased on return.

For example:

1. \_\_weak

Take care when using \_\_weak on the stack. Consider the following example:

NSString\* \_\_weak string = [[NSString alloc] initWithFormat:@"First Name: %@", [self firstName]];

NSLog(@"string: %@", string);

Although *string* is used after the initial assignment, there is no other strong reference to the *string* object at the time of assignment. Therefore, it’s immediately deallocated. The log statement shows that string has a null value. (The compiler provides a warning in this situation.)

1. \_\_strong and \_\_autoreleasing

You need to take care with objects **passed by reference**. The following code will work:

NSError\* error;

BOOL OK = [myObject performOperationWithError:&error];

if (!OK) {

// Report the error.

}

and the method declaration would typically be:

-(BOOL)performOperationWithError:(NSError\* \_\_autoreleasing \*)error;

The compiler therefore rewrites the code:

NSError\* \_\_strong error;

NSError\* \_\_autoreleasing tmp = error;

BOOL OK = [myObject performOperationWithError:&tmp];

error = tmp;

if (!OK) {

// Report the error.

}

The mismatch between the local variable declaration (\_\_strong) and the parameter (\_\_autoreleasing) causes the compiler to create the temporary variable. You can get the original pointer by declaring the parameter id \_\_strong \* when you take the address of a \_\_strong variable. Alternatively, you can declare the variable as \_\_autoreleasing.

Should read [this explanation](https://stackoverflow.com/questions/14554121/what-are-the-advantages-of-declaring-method-arguments-autoreleasing) or [this](https://stackoverflow.com/questions/8862023/in-which-situations-do-we-need-to-write-the-autoreleasing-ownership-qualifier?rq=1) on stackoverflow.

**Note**: When a **variable goes out of scope** (bracket, etc.), ARC automatically decrease the retain count of the referenced instance.

## How Do Blocks Work in ARC?

For what is a block in Objective-C, read [this section](#_37m2jsg).

Blocks “just work” when you pass blocks up the stack in ARC mode, such as in a return. You don’t have to call Block Copy anymore.

The one thing to be aware of is that NSString \* \_\_block myString is retained in ARC mode, not a possibly dangling pointer. To get the previous behavior, use \_\_block NSString \* \_\_weak myString.

# Memory Leaks and Crashes

**Memory Leaks**

### Definition

Leaked memory:

* Inaccessible – no more pointer to it
* Can’t never be used again

Abandoned memory:

* Still reference, but wasted
* Won’t never be used again

Cached memory:

* Referenced and waiting

### Causes

#### Retain Counts > 0

The dealloc is called when an object’s retain count drops to 0 (not necessarily when you release it - other objects may retain it). In other words, **your object is never dealloced because its retain count never drops to 0**. You have a memory leak!

#### Retain Cycles

Retain cycle happens when two objects keep strong references to each other. For [example](https://stackoverflow.com/a/46890898), objectA retains objectB and objectB retains objectA, making it impossible to release.

The simplest method to identify whether there is retain cycle is checking if the dealloc method is called. It it’s not called, there is retain cycle(s).

**IMPORTANT:**

**Fact**: Note that **when an application terminates, objects may not be sent a dealloc message** since the process’s memory is automatically cleared on exit. It is more efficient simply to allow the OS to clean up resources than to invoke all the memory management methods. For this and other reasons, you should not manage scarce resources in dealloc.

**Issue**: The OS will not automatically release "resources" which it can't control (network resources, file handles, etc.) when your application exits. In this case, if you wait for dealloc to clean thing up, dealloc may never come.

*Evidence 1:* Go to Xcode -> Help -> Documentation -> search for NSObject -> and dealloc.

The comment’s point is that the framework is not going to sit there and dealloc every object in your app when the whole process is going away anyway. So **if you have something you absolutely need to clean up in an object which lives until the end of the application lifecycle, then make sure you take care of it in** [**applicationWillTerminate**](https://developer.apple.com/documentation/uikit/uiapplicationdelegate/1623111-applicationwillterminate) (a delegate method on NSApp).

*Evidence 2:* (From Apple) The issue of responsibility for nib object disposal becomes clearer when you consider the various kinds of applications. Most Cocoa applications are of two kinds: single window applications and document-based applications. In both cases, memory management of nib objects is automatically handled for you to some degree. With single-window applications, objects in the main nib file persist through the runtime life of the application and are released when the application terminates; however, dealloc is not guaranteed to be automatically invoked on objects from the main nib file when an application terminates.

*Evidence 3:* The article says that NSWindowController will release nib-instantiated objects. NSApplication won’t. You should not instantiate objects in nibs where FilesOwner isn’t a NSWindowController, NSDocument (which uses a NSWindowController behind the scenes), or a custom class of your own making that takes care of this. Instantiating something in a nib is no different than instantiating it in code. It’s still you who instantiated it, and it’s still you who is responsible for releasing it.

**NOTE:**

Same old note as above. You should never call dealloc directly and you don't have control of when or if it is called. Your job is to provide a complete dealloc method that frees memory you have allocated. The framework's job is to use that dealloc when appropriate.

### Solutions

#### Avoid Retain Cycles

A child should never retain a parent. If anything, use a weak reference in the child to maintain a reference to the parent.

<https://www.cocoawithlove.com/2009/07/rules-to-avoid-retain-cycles.html>

<https://medium.com/@aliakhtar_16369/all-about-memory-leaks-in-ios-cdd450d0cc34>

Others:

Memory Leak in Closure = self refers to → object refers to → self

#### Isolate Possible Problem Areas

If you have been debugging your code and are not sure what is causing your memory problem, it can be helpful to **isolate pieces of code you are suspicious of**. For example, you can temporarily set up your code to call a particular method many times and see how your memory allocations and deallocations react. Also, in a navigation-based app, it’s helpful to continually push and pop the view controller with a potential problem method and see if everything you expect is being released on each pop.

At one point in our app, we weren’t sure if an image-cropping view was causing our app’s memory to climb, so I isolated the whole view controller inside a new Xcode project. After analyzing the project in Xcode Instruments, I saw nothing that would lead me to believe this viewcontroller was the issue. I performed this practice throughout my debugging, and it was effective in showing where the memory problem was not, saving me a lot of time in meticulous debugging.

#### Profiling with Allocations and Leaks Tools (Xcode Instruments)

**Causes of Crashed Due To Poor Memory Management**

### "Zombie"

You have Obj. When its retain count reaches 0, it gets deallocated immediately. It’s now a "zombie" (a pointer to a deallocated object.) If you try to do anything with Obj, you may crash, or the system may store a different object in that memory and your code may actually point to the completely wrong object.

# Strings

// Objective-C

NSString\* s1 = @"hello"; // string literal

NSString\* s2 = [NSString stringWithUTF8String:"A C String"];

sprintf(buff, "%s hello to %@", "there", s2);

const char\* s3 = [s2 UTF8String];

[NSString](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSString_Class/Reference/NSString.html) is a non-mutable representation of an Objective-C string. You can create it by using one of its static methods, or by a string literal with the @ prefix.

**NSString and Encodings**

NSString supports diﬀerent encodings (ASCII, Unicode, ISO Latin 1, etc.), making translation and localization of applications really simple.

**NSString to C String**

The printf function of C has not been extended to support NSString objects. NSLog can be used instead, or any other function that accepts a formatted-string like printf. For an NSString object, the format to use is not “%s” but “%@”. And more generally, “%@” can be used on any object because it is in fact using the returned value of a call to -(NSString\*) description.

Moreover, an NSString can be converted into a C-like string with the UTF8String method

char\* name = "David";

NSString\* action = @"running";

printf("My name is %s, I like %s\n", name, [action UTF8String]);

**Strings Under ARC**

Question:

\_\_strong NSString \*yourString = @"Your String";

\_\_weak NSString \*myString = yourString;

yourString = nil;

I'm expecting myString to be nil at the end, but it’s not. Why?

Answer:

"Your String" is a string literal which will never change (immutable). It technically **won't be released**. Also, retaining a string literal does nothing because it's permanent. **Only nonliteral strings are destroyed like other objects**.

Suggestion:

* Using [[NSString alloc] initWithString:@"literal string"] won't make a difference. Since it becomes a pointer to the literal string. It is however worth noting it works differently and will the release its string object on which it is called.
* Instead, you should initilize your string literals with initWithUTF8String. It will be released under ARC.
* What about initWithFormat and copy?

# Arrays

// Objective-C

NSArray\* a1 = [NSArray alloc] initWithObjects: @"hello", @"there", nil];

NSString\* first = [a1 objectAtIndex:0];

[NSArray](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSArray_Class/NSArray.html) and [NSMutableArray](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSMutableArray_Class/Reference/Reference.html) are the two classes that handle arrays in Objective-C.

The difference is that NSArray elements must be put in construction time by using one of its constructors, whereas NSMutableArray can be altered later. Constructors have the typical form of printf signature, you have to pass nil to "end the items".

There are sort/search/insert functions to NSArray and NSMutableArray as well. In the first case, it returns a new NSArray, whereas in the NSMutableArray case, it modifies the existing object.

# Class Declaration + Access Modifiers

|  |  |
| --- | --- |
| **C++** | **Objective-C** |
| **// In Foo.h file**  class Bar;  class Foo : public CppObject  {  // default = private  int a;  public:  Bar\* p;  void func();  }  **// In Foo.cpp file**  #include "Foo.h"  #include "Bar.h"  void Foo::func()  {  a = 5; // ok  this->a = 5; // ok  CppObject::anyMethod(); // ok  } | **// In Foo.h file**  @class Bar;  @interface Foo : NSObject  {  // default = @protected  int a;  @public  Bar\* p;  }  // methods must be public  - (void)func;  @end  **// In Foo.m file**  #import "Foo.h"  #import "Bar.h"  @implementation Foo  - (void)func  {  a = 5; // ok  self->a = 5; // ok  super.anyMethod(); // parent call  }  @end |

Differences:

* In Objective-C, implementation scope is within the @implementation/@end tags (where in C++ we can implement it anywhere using the :: scope operator).
* Objective-C uses the @class keyword for forward declarations.
* Unlike C++, methods in Objective-C must be public.
* Instance variables (member variables) can be public, private or protected.
* Objective-C uses selfinstead of this (in C++) and it calls its parent class through the superkeyword.
* Inheritance cannot be public, protected or private. It is only public.

# Inheritance

## Single Inheritance

In C++, a class can be derived from another in public, protected or private mode. In the methods, one can reference a superclass using the scope resolution operator :: (Bar::, Wiz::).

In Objective-C, a class can derive from another in only public mode. A method can reference the superclass using keyword super.

NOTE: Objective-C does not implement **multiple inheritance**.

## Protocol

In Objective-C, protocol is the alternative of **interface** in C++. And it is allowed to contain data.

|  |  |
| --- | --- |
| **In C++** | **In Objective-C** |
| class MouseListener  {  public:  virtual bool mousePressed(void) = 0;  virtual bool mouseClicked(void) = 0;  };  class KeyboardListener  {  public:  virtual bool keyPressed(void) = 0;  };  class Foo : public MouseListener,  public KeyboardListener {...}  // Foo MUST implement mousePressed, mouseClicked and keyPressed | @protocol MouseListener  -(BOOL) mousePressed;  -(BOOL) mouseClicked;  @end  @protocol KeyboardListener  -(BOOL) keyPressed;  @end  @interface Foo : NSObject <MouseListener, KeyboardListener> { ... }  @end  // Foo MUST implement mousePressed, mouseClicked and keyPressed |

**Conform to Protocol**

// Objective-C

NSObject\* ptr = ...;

if ([ptr conformsToProtocol:@protocol(Foo)] {

[ptr func:5:3];

}

// C++

CppObject\* ptr = ...; // some pointer that also inherits from foo

Foo\* f = ptr; // or, the compiler warns that ptr isn't compatible with foo.

f->fuc(5,3);

Now we check if the receiver **conforms to a protocol**(or, in C++, implements an interface), so we can send messages that this protocol contains. Hey, it's mostly like Java classes and interfaces, where, in C++, there is no technical difference between a fully implemented class and an "interface".

**@optional and @required**

4.4.2 From C++ to Objective-C

**Informal Protocol**

4.4.3 From C++ to Objective-C

**Keywords: in, out, inout, bycopy, byref and oneway**

4.4.5 From C++ to Objective-C

## Virutal Methods

In Objective-C, all methods are virtual. Hence, the virtual keyword does not exist and has no equivalent.

## Silent Redefinition

4.3.2 From C++ to Objective-C

# Categories

## Why Use Categories?

Use case #1. You typically use a category to **add methods or properties (NOT instance variables) to an existing class, such as one defined in the Cocoa frameworks, without subclassing**. The **added methods or properties are inherited by subclasses** and are indistinguishable at runtime from the original methods or properties of the class.

Use case #2. **Distribute the implementation of your own classes into separate source files**. For example, you could group the methods of a large class into several categories and put each category in a different file.

Use case #3. **Declare private methods or properties**. You add methods or properties to a class by declaring them in an interface file under a category name and defining them in an implementation file under the same name. The category name indicates that the methods are an extension to a class declared elsewhere, not a new class.

## Declaration

For use case #1+2:

The declaration of a category interface looks very much like a class interface – except that the category name is **listed within parentheses after the class** **name** and the superclass isn’t mentioned.

*Naming convention*: The file name of the category is the name of the class the category extends followed by “+” followed by the name of the category. For example, *SystemClass+SystemCategory.h*.

// In SystemClass+SystemCategory.h

#import "SystemClass.h"

@interface SystemClass (SystemCategory)

// No instance variable can be added inside a category, so there is no {...} block.

// method or properties declarations

@end

For use case #3:

If you use a category to declare **private methods** of one of your own classes, you can put the declaration in the implementation file before the @implementation block:

#import "MyClass.h"

@interface MyClass (MyCategory)

// No instance variable can be added inside a category, so there is no {...} block.

// method or properties declarations

@end

@implementation MyClass

// method or properties definitions

@end

## Implementation

For use case #1+2: Put the implementation in a file named *ClassName+CategoryName.m*

For use case #3: Put the implementation in your class’s @implementation block

// In SystemClass+SystemCategory.m

#import "SystemClass+SystemCategory.h"

@implementation SystemClass (SystemCategory)

// method or properties definitions

@end

## One More Usage…

|  |  |
| --- | --- |
| **In C++** | **In Objective-C** |
| #include <string>  #include "MyString.h"  class MyString : public string  {  public:  void printString() {  printf("%s", c\_str());  }  };  MyString s1 = "hi";  s1.printString(); // ok  string s2 = "hello";  s2.printString(); // error, we must change s2 from string to MyString | @interface NSString (MyString)  - (void) printString;  @end  @implementation NSString (MyString)  - (void) printString {  printf("%@", self);  }  @end  NSString\* s2 = @"hello";  [s2 printString]; // ok. We extended NSString without changing types. |

C++ relies on **inheritance**to extend a known class. This can be bothersome because all users of an extended class must use another type (in that case, MyString instead of string).

With categories in Objective-C, we don’t need to change type of the extended class. This allows all source files that view the extension .h file to instantly become able to call the new member functions without the need to cast NSString to MyString.

## Anonymous Categories

If you don’t want to name the categories you created, you can leave the parentheses empty. For example:

@interface SystemClass ()

...

@end

// Or

@interface MyClass ()

...

@end

# Downcasting

Downcasting (using dynamic\_cast) is needed in C to call methods on a subclass, when only a parent class pointer is known. In Objective-C, this practice is not necessary because a message can be sent to an object event if it seems that the object cannot handle it.

However, **to avoid a compiler warning, we should doown-cast the type of the object**. There is no explicit downcasting operator in Objective-C.

Ex1:

|  |  |
| --- | --- |
| **In C++** | **In Objective-C** |
| CppObject\* ptr = ...;  Foo\* f = dynamic\_cast<Foo\*>(ptr);  if (f) {  f->func(5,3);  } | NSObject\* ptr = ...;  if ([ptr isKindOfClass:[Foo class]] {  [ptr func:5:3];  } |

Ex2:

// NSMutableString is a subclass of NSString that allows mutating operations

// the "appendString:" method only exists in NSMutableString

NSMutableString\* mutableString = ... // initialize a mutable string

NSString\* string = mutableString; // store in an NSString pointer

// Below different calls are all valid

[string appendString:@"foo"]; // compiler warning

[(NSMutableString\*)string appendString:@"foo"]; // no warning

[(id)string appendString:@"foo"]; // no warning

# Blocks

<https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithBlocks/WorkingwithBlocks.html>

<https://stackoverflow.com/questions/11773342/what-is-the-difference-between-a-weak-and-a-block-reference>

<https://spin.atomicobject.com/2017/01/22/avoiding-objective-c-memory-leaks/>

<https://holko.pl/2015/05/31/weakify-strongify/>

# C++ and Objective-C

## Casting: Non-Objective-C Objects <--> Objective-C Objects

* Non-Objective-C objects are NOT managed by ARC. For example: **C**ore **F**oundation objects (**CF**StringRef, etc.) or **C**ore **G**raphics (**CG**ColorSpaceRef, etc.).
* Objective-C objects are managed by ARC. For example: Foundation objects (NSString\*, etc.).

### Non-Objective-C Object --> Objective-C Object

Ex:

// Create a string of type ‘CFStringRef’ - it is a non-Objective-C pointer

CFStringRef cfStr = CFString**Create**Copy(kCFAllocatorDefault, CFSTR("xyz"));

// Cast CFStringRef to NSString\* and also transfer its ownership to ARC

NSString\* nsStr = (\_\_bridge\_transfer NSString\*)cfStr;

// Or

// NSString\* nsStr = (NSString\*)**CFBridgingRelease**(cfStr);

// Because the ownership is now transferred to ARC, NO need to manually release

// CFRelease(cfStr);

Note: Using \_\_bridge does cast CFStringRef to NSString\*, but it doesn’t transfer the string’s ownership to ARC. So, we need to manually release CFStringRef.

CFStringRef cfStr = CFStringCreateCopy(kCFAllocatorDefault, CFSTR("xyz"));

NSString\* nsStr = (\_\_bridge NSString\*)cfStr;

CFRelease(cfStr);

### Objective-C Object --> Non-Objective-C Object

Ex:

NSString\* nsStr = ...;

// Cast NSString\* to CFStringRef

CFStringRef\* cfStr = (\_\_bridge CFStringRef\*)nsStr;

// Do NOT need to release CFStringRef after using

// CFRelease(cfStr);

Note: Only when you want to manage the lifecycle of the casted CFStringRef, you use \_\_bridge\_retained as follow:

NSString\* nsStr = ...;

CFStringRef\* cfStr = (\_\_bridge\_retained CFStringRef\*)nsStr;

// Or

// CFStringRef cfStr = (CFStringRef)**CFBridgingRetain**(nsStr);

// Some long time later, perhaps in another method

CFRelease(cfStr);

// So the \_\_bridge\_retained is telling the compiler that I had an ARC object and now I want to basically turn it into a CF object that I’m going to manage directly.

// Note: This solution may cause memory leaks! CANNOT UNDERSTAND!

## When To Implement Dealloc Under ARC?

Generally, ARC will automatically generate appropriate dealloc method to free all Objective-C objects. So we do not need to override and implement this method under ARC when working with only Objective-C objects

However, ARC does NOT automatically count references on C++ objects. So we have to implement dealloc which contains only implementation for releasing C++ objects.

## Release vs Debug

// C++

class A

{

public:

NSObject\* s;

A();

};

A::A()

{

s = 0; // Might boom, usually in release mode!

}

Let's see what happens here. The s = 0 line assigns 0 to the variable, and therefore, whatever was in that variable before must be released first, so the compiler executes a [s release] before assignment. If s was already 0, as in **debug builds,** nothing happens; it's perfectly valid to use [s release] if s is nil. However, in release builds, s might be a dangling pointer, so it may contain any value before its "initialization" to 0.

In C++, this is not a problem because no ARC is there. In Objective-C, however, the compiler does not have a way to know if this is an "assignment" or "initialization" (if the latter case, it will not send the release message.

This is the correct approach:

// C++

class A

{

public:

NSObject\* s;

A();

};

// now the compiler knows for sure it's an initialization, so no [s release]

A::A() :s(0)

{

}

Now the compiler won't try a [s release] because it knows it's the first initialization of the object. Be careful!

**Others**

* Cannot use property attributes and variable qualifiers for objects created by C++ (eg: '\_\_weak char\*' is invalid.). The compiler will give warnings.

# Summary

What Objective-C has and C++ hasn't:

* Categories
* NSObject-based operations
* YES and NO (equal to true and false)
* NIL and nil (equal to 0)
* Named function arguments
* self (equal to this) but it can be changed in a constructor

What C++ has and Objective-C hasn't:

* Static objects (Objects in Objective-C cannot be instantiated statically or in the stack. Only pointers)
* References (&) do not exist in Objective-C.
* Multiple inheritance
* Namespaces
* Templates
* Operator overloading
* STL and algorithms
* protected or private methods (In Objective-C, all methods are public)
* const methods (In Objective-C, methods cannot be set to const)
* virtual methods (In Objective-C, all methods are virtual. Pure virtual methods are implemented with a formal protocol)
* friend methods
* throw keyword (Objective-C has try-catch, but it doesn’t allow to restrict a method to transmit some exceptions only like the way C++ does)
* References
* Anonymous function signatures (without a variable name)