Blocks Programming Topics

Languages

2009-10-19

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Introduction

Block objects are a C-level syntactic and runtime feature. They are similar to standard C functions, but in addition to executable code they may also contain variable bindings to automatic (stack) or managed (heap) memory. Blocks allow you to compose function expressions that that can be passed to API, optionally stored, and used by multiple threads.

Blocks are available in GCC and Clang as shipped with the Mac OS X v10.6 Xcode developer tools. The blocks runtime is open source and can be found in LLVM's compiler-rt subproject repository. Blocks have also been presented to the C standards working group as N1370: Apple's Extensions to C (which also includes Garbage Collection). As Objective-C and C++ are both derived from C, blocks are designed to work with all three languages (as well as Objective-C++). (The syntax reflects this goal).

You should read this document to learn what block objects are and how you can use them from C, C++, or Objective-C to make your program more efficient and more maintainable.

Organization of This Document

This document contains the following chapters:

- "Getting Started with Blocks" (page 7) provides a quick, practical, introduction to blocks.
- "Conceptual Overview" (page 9) provides a conceptual introduction to blocks.
- "Declaring and Creating Blocks" (page 11) shows you how to declare block variables and how to implement blocks.
- "Blocks and Variables" (page 13) describes the interaction between blocks and variables, and defines the the `__block` storage type modifier.
- "Using Blocks" (page 17) illustrates various usage patterns.

Getting Started with Blocks

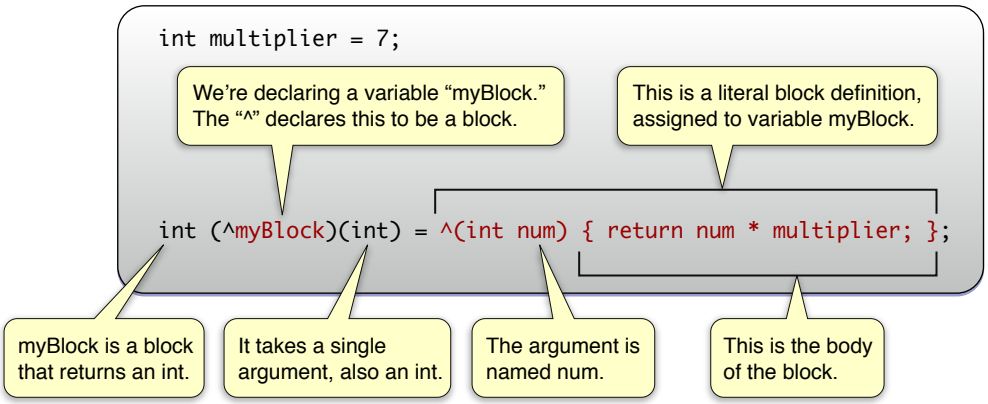
The following sections help you to get started with blocks using practical examples.

Declaring and Using a Block

You use the ^ operator to declare a block variable and to indicate the beginning of a block literal (as usual with C, ; indicates the end of the expression), as shown in this example:

```
int multiplier = 7;
int (^myBlock)(int) = ^(int num) {
    return num * multiplier;
};
```

The example is explained in the following illustration:



Notice that the block is able to make use of variables from the same scope in which it was defined.

If you declare a block as a variable, you can then use it just as you would a function:

```
int multiplier = 7;
int (^myBlock)(int) = ^(int num) {
    return num * multiplier;
};

printf("%d", myBlock(3));
// prints "21"
```

Using a Block Directly

In many cases, you don't need to declare block variables; instead you simply write a block literal inline where it's required as an argument. The following example uses the `qsort_b` function. `qsort_b` is similar to the standard `qsort_r` function, but takes a block as its final argument.

```
char *myCharacters[3] = { "TomJohn", "George", "Charles Condomine" };

qsort_b(myCharacters, 3, sizeof(char *), ^(const void *l, const void *r) {
    char *left = *(char **)l;
    char *right = *(char **)r;
    return strncmp(left, right, 1);
});

// myCharacters is now { "Charles Condomine", "George", TomJohn" }
```

Blocks with Cocoa

Several methods in the Cocoa frameworks take a block as an argument, typically either to perform an operation on a collection of objects, or to use as a callback after an operation has finished. The following example shows how to use a block with the `NSArray` method `sortedArrayUsingComparator:`. The method takes a single argument—the block. In this case the block is defined as an `NSComparator`:

```
NSArray *stringsArray = [NSArray arrayWithObjects:
                                  @"string 1",
                                  @"String 21",
                                  @"string 12",
                                  @"String 11",
                                  @"String 02", nil];

static NSStringCompareOptions comparisonOptions = NSCaseInsensitiveSearch |
NSNumericSearch |
        NSWidthInsensitiveSearch | NSForcedOrderingSearch;
NSLocale *currentLocale = [NSLocale currentLocale];

NSComparator finderSort = ^(id string1, id string2) {

    NSRange string1Range = NSMakeRange(0, [string1 length]);
   return [string1 compare:string2 options:comparisonOptions range:string1Range
 locale:currentLocale];
};

NSLog(@"finderSort: %@", [stringsArray sortedArrayUsingComparator:finderSort]);

/*
Output:
finderSort: (
    "string 1",
    "String 02",
    "String 11",
    "string 12",
    "String 21"
)
```

Conceptual Overview

Block objects provide a way for you to create an ad hoc function body as an expression in C, and C-derived languages such as Objective-C and C++. In other languages and environments, a block object is sometimes also called a "closure". Here, they are typically referred to colloquially as "blocks", unless there is scope for confusion with the standard C term for a block of code.

Block Functionality

A block is an anonymous inline collection of code that:

- Has a typed argument list just like a function
- Has an inferred or declared return type
- Can capture state from the lexical scope within which it is defined
- Can optionally modify the state of the lexical scope
- Can share the potential for modification with other blocks defined within the same lexical scope
- Can continue to share and modify state defined within the lexical scope (the stack frame) after the lexical scope (the stack frame) has been destroyed

You can copy a block and even pass it to other threads for deferred execution (or, within its own thread, to a runloop). The compiler and runtime arrange that all variables referenced from the block are preserved for the life of all copies of the block. Although blocks are available to pure C and C++, a block is also always an Objective-C object.

Usage

Blocks represent typically small, self-contained pieces of code. As such, they're particularly useful as a means of encapsulating units of work that may be executed concurrently, or over items in a collection.

Because of their interaction with variables, blocks are also useful in situations where you need access to local state. Blocks are therefore also a useful alternative to callbacks. Rather than using callbacks requiring a data structure that embodies all the contextual information you need to perform an operation, you simply access local variables directly.

Declaring and Creating Blocks

Declaring a Block Reference

Block variables hold references to blocks. You declare them using syntax similar to that you use to declare a pointer to a function, except that you use ^ instead of *. The block type fully interoperates with the rest of the C type system. The following are all valid block variable declarations:

```
void (^blockReturningVoidWithVoidArgument)(void);
int (^blockReturningIntWithIntAndCharArguments)(int, char);
void (^arrayOfTenBlocksReturningVoidWithIntArgument[10])(int);
```

Blocks also support variadic (. . .) arguments. A block that takes no arguments must specify `void` in the argument list.

Blocks are designed to be fully type safe by giving the compiler a full set of metadata to use to validate use of blocks, parameters passed to blocks, and assignment of the return value. You can cast a block reference to a pointer of arbitrary type and vice versa. You cannot, however, dereference a block reference via the pointer dereference operator (*)—a block's size cannot be computed at compile time.

You can also create types for blocks—doing so is generally considered to be best practice when you use a block with a given signature in multiple places:

```
typedef float (^MyBlockType)(float, float);

MyBlockType myFirstBlock = // ... ;
MyBlockType mySecondBlock = // ... ;
```

Creating a Block

You use the ^ operator to indicate the beginning of a block literal expression and ; to indicate the end of a block expression. The following example declares a simple block and assigns it to a previously declared variable (`oneFrom`):

```
int (^oneFrom)(int);

oneFrom = ^(int anInt) {
    return anInt - 1;
};
```

If you don't explicitly declare the return value of a block expression, it can be automatically inferred from the contents of the block. If the return type is inferred and the parameter list is `void`, then you can omit the `(void)` parameter list as well. If or when multiple return statements are present, they must exactly match (using casting if necessary).

Global Blocks

At a file level, you can use a block as a global literal:

```
#import <stdio.h>

int GlobalInt = 0;
int (^getGlobalInt)(void) = ^{ return GlobalInt; };
```

Blocks and Variables

This article describes the interaction between blocks and variables, including memory management.

Types of Variable

Within the block object's body of code, variables may be treated in five different ways.

You can reference three standard types of variable, just as you would from a function:

- Global variables, including static locals
- Global functions (which aren't technically variables)
- Local variables and parameters from an enclosing scope

Blocks also support two other types of variable:

1. At function level are `__block` variables. These are mutable within the block (and the enclosing scope) and are preserved if any referencing block is copied to the heap.
2. `const` imports.

Finally, within a method implementation, blocks may reference Objective-C instance variables—see "Object and Block Variables" (page 15).

The following rules apply to variables used within a block:

1. Global variables are accessible, including static variables that exist within the enclosing lexical scope.
2. Parameters passed to the block are accessible (just like parameters to a function).
3. Stack (non-static) variables local to the enclosing lexical scope are captured as `const` variables.

   Their values are taken at the point of the block expression within the program. In nested blocks, the value is captured from the nearest enclosing scope.
4. Variables local to the enclosing lexical scope declared with the `__block` storage modifier are provided by reference and so are mutable.

   Any changes are reflected in the enclosing lexical scope, including any other blocks defined within the same enclosing lexical scope. These are discussed in more detail in "The __block Storage Type" (page 14).
5. Local variables declared within the lexical scope of the block, which behave exactly like local variables in a function.

Each invocation of the block provides a new copy of that variable. These variables can in turn be used as `const` or by-reference variables in blocks enclosed within the block.

The following example illustrates the use of local non-static variables:

```
int x = 123;

void (^printXAndY)(int) = ^(int y) {

    printf("%d %d\n", x, y);
};

printXAndY(456); // prints: 123 456
```

As noted, trying to assign a new value to `x` within the block would result in an error:

```
int x = 123;

void (^printXAndY)(int) = ^(int y) {

    x = x + y; // error
    printf("%d %d\n", x, y);
};
```

To allow a variable to be changed within a block, you use the `__block` storage type modifier—see "The __block Storage Type" (page 14).

The __block Storage Type

You can specify that an imported variable be mutable—that is, read-write— by applying the `__block` storage type modifier. `__block` storage is similar to, but mutually exclusive of, the `register`, `auto`, and `static` storage types for local variables.

`__block` variables live in storage that is shared between the lexical scope of the variable and all blocks and block copies declared or created within the variable's lexical scope. Thus, the storage will survive the destruction of the stack frame if any copies of the blocks declared within the frame survive beyond the end of the frame (for example, by being enqueued somewhere for later execution). Multiple blocks in a given lexical scope can simultaneously use a shared variable.

As an optimization, block storage starts out on the stack—just like blocks themselves do. If the block is copied using `Block_copy` (or in Objective-C when the block is sent a `copy`), variables are copied to the heap. Thus, *the address of a* `__block` *variable can change over time*.

There are two further restrictions on `__block` variables: they cannot be variable length arrays, and cannot be structures that contain C99 variable-length arrays.

The following example illustrates use of a `__block` variable:

```
__block int x = 123; //  x lives in block storage

void (^printXAndY)(int) = ^(int y) {

    x = x + y;
```

```
    printf("%d %d\n", x, y);
};
printXAndY(456); // prints: 579 456
// x is now 579
```

The following example shows the interaction of blocks with several types of variable:

```
extern NSInteger CounterGlobal;
static NSInteger CounterStatic;

{
    NSInteger localCounter = 42;
    __block char localCharacter;

    void (^aBlock)(void) = ^(void) {
        ++CounterGlobal;
        ++CounterStatic;
        CounterGlobal = localCounter; // localCounter fixed at block creation
        localCharacter = 'a'; // sets localCharacter in enclosing scope
    };

    ++localCounter; // unseen by the block
    localCharacter = 'b';

    aBlock(); // execute the block
    // localCharacter now 'a'
}
```

Object and Block Variables

Blocks provide support for Objective-C and C++ objects, and other blocks, as variables.

Objective-C Objects

In a reference-counted environment, by default when you reference an Objective-C object within a block, it is retained. This is true even if you simply reference an instance variable of the object. Object variables marked with the `__block` storage type modifier, however, are not retained.

Note: In a garbage-collected environment, if you apply both `__weak` and `__block` modifiers to a variable, then the block will not ensure that it is kept alive.

If you use a block within the implementation of a method, the rules for memory management of object instance variables are more subtle:

- If you access an instance variable by reference, `self` is retained;
- If you access an instance variable by value, the variable is retained.

The following examples illustrate the two different situations:

```
dispatch_async(queue, ^{
```

```
    // instanceVariable is used by reference, self is retained
    doSomethingWithObject(instanceVariable);
});


id localVariable = instanceVariable;
dispatch_async(queue, ^{
    // localVariable is used by value, localVariable is retained (not self)
    doSomethingWithObject(localVariable);
});
```

C++ Objects

In general you can use C++ objects within a block. Within a member function, references to member variables and functions are via an implicitly imported `this` pointer and thus appear mutable. There are two considerations that apply if a block is copied:

- If you have a `__block` storage class for what would have been a stack-based C++ object, then the usual `copy` constructor is used.
- If you use any other C++ stack-based object from within a block, it must have a `const copy` constructor. The C++ object is then copied using that constructor.

Blocks

When you copy a block, any references to other blocks from within that block are copied if necessary—an entire tree may be copied (from the top). If you have block variables and you reference a block from with the block, that block will be copied.

When you copy a stack-based block, you get a new block. If you copy a heap-based block, however, you simply increment the retain count of that block and get it back as the returned value of the copy function or method.

Using Blocks

Invoking a Block

If you declare a block as a variable, you can use it as you would a function, as shown in these two examples:

```
int (^oneFrom)(int) = ^(int anInt) {
    return anInt - 1;
};

printf("1 from 10 is %d", oneFrom(10));
// Prints "1 from 10 is 9"

float (^distanceTraveled) (float, float, float) =
                          ^(float startingSpeed, float acceleration, float time)
 {

    float distance = (startingSpeed * time) + (0.5 * acceleration * time * time);
    return distance;
};

float howFar = distanceTraveled(0.0, 9.8, 1.0);
// howFar = 4.9
```

Frequently, however, you pass a block as the argument to a function or a method. In these cases, you usually create a block "inline".

Using a Block as a Function Argument

You can pass a block as function argument just as you would any other argument. In many cases, however, you don't need to declare blocks; instead you simply implement them inline where they're required as an argument. The following example uses the `qsort_b` function. `qsort_b` is similar to the standard `qsort_r` function, but takes a block as its final argument.

```
char *myCharacters[3] = { "TomJohn", "George", "Charles Condomine" };

qsort_b(myCharacters, 3, sizeof(char *), ^(const void *l, const void *r) {
    char *left = *(char **)l;
    char *right = *(char **)r;
    return strncmp(left, right, 1);
});
// Block implementation ends at "}"

// myCharacters is now { "Charles Condomine", "George", TomJohn" }
```

Notice that the block is contained within the function's argument list.

The next example shows how to use a block with the `dispatch_apply` function. `dispatch_apply` is declared as follows:

```
void dispatch_apply(size_t iterations, dispatch_queue_t queue, void
(^block)(size_t));
```

The function submits a block to a dispatch queue for multiple invocations. It takes three arguments; the first specifies the number of iterations to perform; the second specifies a queue to which the block is submitted; and the third is the block itself, which in turn takes a single argument—the current index of the iteration.

You can use `dispatch_apply` trivially just to print out the iteration index, as shown:

```
#include <dispatch/dispatch.h>
size_t count = 10;
dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_apply(count, queue, ^(size_t i) {
    printf("%u\n", i);
});
```

Using a Block as a Method Argument

Cocoa's Foundation framework provides a number of methods that use blocks. You pass a block as method argument just as you would any other argument.

The following example determines the indexes of any of the first five elements in an array that appear in a given filter set.

```
NSArray *array = [NSArray arrayWithObjects: @"A", @"B", @"C", @"A", @"B",
@"Z",@"G", @"are", @"Q", nil];
NSSet *filterSet = [NSSet setWithObjects: @"A", @"Z", @"Q", nil];

BOOL (^test)(id obj, NSUInteger idx, BOOL *stop);

test = ^ (id obj, NSUInteger idx, BOOL *stop) {

    if (idx < 5) {
        if ([filterSet containsObject: obj]) {
            return YES;
        }
    }
    return NO;
};

NSIndexSet *indexes = [array indexesOfObjectsPassingTest:test];

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

/*
Output:
indexes: <NSIndexSet: 0x10236f0>[number of indexes: 2 (in 2 ranges), indexes:
(0 3)]
*/
```

The following example determines whether an `NSSet` object contains a word specified by a local variable and sets the value of another local variable (`found`) to `YES` (and stops the search) if it does. Notice that `found` is also declared as a `__block` variable, and that the block is defined inline:

```
__block BOOL found = NO;
NSSet *aSet = [NSSet setWithObjects: @"Alpha", @"Beta", @"Gamma", @"X", nil];
NSString *string = @"gamma";

[aSet enumerateObjectsUsingBlock:^(id obj, BOOL *stop) {
    if ([obj localizedCaseInsensitiveCompare:string] == NSOrderedSame) {
        *stop = YES;
        found = YES;
    }
}];

// At this point, found == YES
```

Copying Blocks

Typically, you shouldn't need to copy (or retain) a block. You only need to make a copy when you expect the block to be used after destruction of the scope within which it was declared. Copying moves a block to the heap.

You can copy and release blocks using C functions:

```
Block_copy();
Block_release();
```

If you are using Objective-C, you can send a block `copy`, `retain`, and `release` (and `autorelease`) messages.

To avoid a memory leak, you must always balance a `Block_copy()` with `Block_release()`. You must balance `copy` or `retain` with `release` (or `autorelease`)—unless in a garbage-collected environment.

Patterns to Avoid

A block literal (that is, `^{ ... }`) is the address of a *stack-local* data structure that represents the block. The scope of the stack-local data structure is therefore the enclosing compound statement, so you should *avoid* the patterns shown in the following examples:

```
void dontDoThis() {
    void (^blockArray[3])(void);  // an array of 3 block references

    for (int i = 0; i < 3; ++i) {
        blockArray[i] = ^{ printf("hello, %d\n", i); };
        // WRONG: The block literal scope is the "for" loop
    }
}

void dontDoThisEither() {
    void (^block)(void);
```

```
    int i = random():
    if (i > 1000) {
        block = ^{ printf("got i at: %d\n", i); };
        // WRONG: The block literal scope is the "then" clause
    }
    // ...
}
```

Debugging

You can set breakpoints and single step into blocks. You can invoke a block from within a GDB session using `invoke-block`, as illustrated in this example:

```
$ invoke-block myblock 10 20
```

If you want to pass in a C string, you must quote it. For example, to pass `this string` into the `doString` block, you would write the following:

```
$ invoke-block doString "\"this string\""
```

Document Revision History

This table describes the changes to *Blocks Programming Topics*.

Date	Notes
2009-10-19	Clarified aspects of memory management and type inferencing.
2009-05-28	New document that describes the Blocks feature for the C programming language.

Document Revision History