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 a 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 provides a number of methods that use blocks. You pass a block as a 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 = @[@"A", @"B", @"C", @"A", @"B", @"Z", @"G", @"are", @"Q"];
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();
```

To avoid a memory leak, you must always balance a `Block_copy()` with `Block_release()`.

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 `doSomethingWithString` block, you would write the following:

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