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.

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.
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.

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 variables:

```
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

When a block is copied, it creates strong references to object variables used within the block. If you use a block within the implementation of a method:

- If you access an instance variable by reference, a strong reference is made to `self`;
- If you access an instance variable by value, a strong reference is made to the variable.

The following examples illustrate the two different situations:

```
dispatch_async(queue, ^{
    // instanceVariable is used by reference, a strong reference is made to self
    doSomethingWithObject(instanceVariable);
});



id localVariable = instanceVariable;
dispatch_async(queue, ^{
    /*
      localVariable is used by value, a strong reference is made to localVariable
      (and not to self).
    */
    doSomethingWithObject(localVariable);
});
```

To override this behavior for a particular object variable, you can mark it with the `__block` storage type modifier.

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 within the block, that block will be copied.

Copyright © 2011 Apple Inc. All Rights Reserved. Terms of Use | Privacy Policy | Updated: 2011-03-08