Blocks and Grand Central Dispatch

Benedikt Meurer

CocoaHeads Siegen

2011/08/10

Blocks and Grand Central Dispatch

- Blocks
  - Blocks in C
  - Blocks in Objective-C
  - Blocks in Cocoa(Touch)
- Grand Central Dispatch

Blocks

Blocks

- Nonstandard extension to the C, C++ and Objective-C/C++ languages by Apple
- Available with clang and Apple's gcc (starting with OS X 10.6 and iOS 4.0)
- Like functions, but written inline with the rest of your code
- Closures or λ-expressions for C

A simple example

```
#include <Block.h>
#include <stdio.h>
typedef int (^IntBlock)();

IntBlock CreateCounter(int start, int increment) {
   __block int c = start;
   return Block_copy(^{
      int result = c;
      c += increment;
      return result;
   });
}

int main(int argc, char *argv[]) {
   IntBlock counter = CreateCounter(7, 2);
   printf("1st: %d\n", counter());
   printf("2nd: %d\n", counter());
   Block_release(counter);
   return 0;
}
```

A simple example

- Compile and run the example

```
$ clang -fblocks example1.c -o example1
$ ./example1
1st: 7
2nd: 9
```

What to do with'em?

- Custom control structures (like in Ruby or functional languages), i.e.

```
[1, 2, 3, 4].each do |i|
  puts i
end
```

```
List.iter
(fun i -> print_int i)
[1; 2; 3; 4]
```

- Callbacks
- Delayed execution
- Building blocks for concurrency

Blocks in C

Blocks in C

- Block types similar to function types

```
double (*funcptr)(int);        double (^blkptr)(int);
typedef int (*FuncType)();     typedef int (^BlkType)();
```

- New syntax for declaring blocks

```
blkvar = ^ type (type arg1, ..., type argn) {
   statements;
   return value;
};
```

- Abbreviations

```
^ type { ... }
```
skip empty argument list

```
^ { ... }
```
infers return type

Calling blocks

- Just like function calls

```
typedef int (^IntBlock)();
IntBlock solutionBlock = ^{
  return 42;
};

int solution = solutionBlock();

int (^add2solutionBlock)(int) = ^int (int x) {
  return solution + x;
};

int solutionPlus7 = add2solution(7);
```

Using variables in closure scope

- Just works for read-only access (no need for Java's `final` qualifier)
- Variables writable from inside closures need `__block` qualifier, i.e.

```
void foreach(List *list, void (^block)(List *)) {
  for (; list; list = list->next) block(list);
}
...
List *list = ...;
__block int count = 0;
foreach(list, ^void(CFTypeRef element) {
  count++;
});
printf("Number of items in list: %d\n", count);
```

The magical __block

```
typedef struct { int (^up)(); int (^down)(); } Counter;
Counter CreateCounter(int start, int inc) {
  __block int i = start;
  Counter c = {
    .up = Block_copy(^{ int r = i; i += inc; return r; }),
    .down = Block_copy(^{ int r = i; i -= inc; return r; })
  };
  return c;
}

int main(int argc, char *argv[]) {
  Counter c = CreateCounter(10, 1);
  printf("1st: %d\n", c.up());
  printf("2nd: %d\n", c.up());
  printf("3rd: %d\n", c.down());
  Block_release(c.up);
  Block_release(c.down);
  return 0;
}
```

The magical __block

Shared state

```
typedef struct { int (^up)(); int (^down)(); } Counter;
Counter CreateCounter(int start, int inc) {
  __block int i = start;
  Counter c = {
    .up = Block_copy(^{ int r = i; i += inc; return r; }),
    .down = Block_copy(^{ int r = i; i -= inc; return r; })
  };
  return c;
}

int main(int argc, char *argv[]) {
  Counter c = CreateCounter(10, 1);
  printf("1st: %d\n", c.up());
  printf("2nd: %d\n", c.up());
  printf("3rd: %d\n", c.down());
  Block_release(c.up);
  Block_release(c.down);
  return 0;
}
```

i survived!

The magical `__block`

- `i` survived it's enclosing scope
- impossible for an automatic (stack)variable
- conclusio: `__block` does some heap-allocation magic
- We'll get to that soon... some basics about block memory management first

Memory Management

- Block consists of code and state
- Block code just like all other code, ends up in `.text` section
- Block state is the variables enclosed and some internal stuff
- Memory for block state must be managed somehow - remember, we're talking C!

Memory Management

- Declaring a block in function scope actually creates a *block literal* on the *stack*

```
int (^one)() = ^{ return 1; };
```

```
// compiles to a separate function
static int one_invoke(struct Block_literal_1 *b) {
  return 1;
}
...
// and the following code in scope
// (see Block-ABI-Apple.txt for types)
struct Block_literal_1 one_storage = {
  .isa = &_NSConcreteStackBlock,
  .invoke = one_invoke,...
};
struct Block_literal_1 *one = &one_storage;
```

Memory Management

- Declaring a block in global scope actually creates a *block literal* in the `.data` section

```
static int (^one)() = ^{ return 1; };
```

```
// compiles to a separate function
static int one_invoke(struct Block_literal_1 *b) {
  return 1;
}
...
// and the following code in global scope
// (see Block-ABI-Apple.txt for types)
static struct Block_literal_1 one_storage = {
  .isa = &_NSConcreteGlobalBlock,
  .invoke = one_invoke,...
};
static struct Block_literal_1 *one = &one_storage;
```

Memory Management

- Stack-allocated block literals are only valid in their declaring scope
- That's why we used `Block_copy()` in the examples

`Block_copy()` and `Block_release()`

- Blocks in local scope are stack-allocated, because:

+ Stack allocation is fast/cheap

+ Deallocation is automatic (for free)

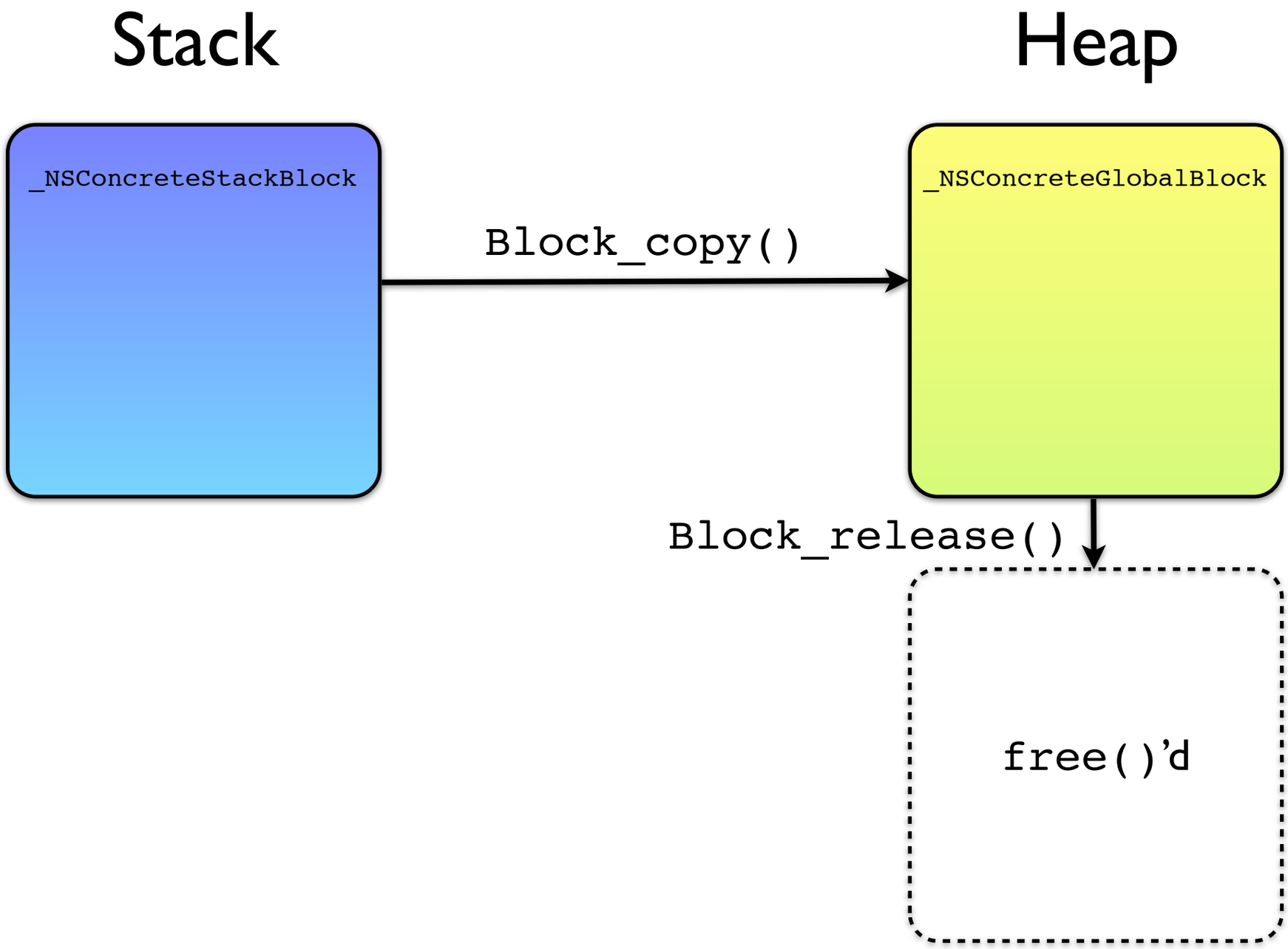
+ Most blocks don't need to survive their declaring scope

- GCC nested functions feature already does this... but Blocks do more!

`Block_copy()` and `Block_release()`

- `Block_copy()` copies a block from stack to heap memory (or retains a block already in heap memory)
- `Block_release()` releases a block in heap memory
- Two distinct classes `_NSConcreteStackBlock` and `_NSConcreteGlobalBlock` to make this explicit

Block_copy() and Block_release()



`Block_copy()` and `Block_release()`

- Blocks start out stack-allocated
- `Block_copy()` to copy to/retain in heap
- `Block_release()` to release in heap
- Rule of thumb: Copy blocks whenever they may survive their *declaring scope*!!

Block_copy() and Block_release()

```
typedef void (^Block)();

Block f() { // obvious bug
  return ^{ ... };
}

void g() { // tricky bug
  Block b;
  if (whatever) {
    b = ^{ ... };
  }
  else {
    b = ^{ ... };
  }
  b(); // called out of block literal scope
}
```

Now what about this `__block` thing?

- Modifier `__block` only valid for variables with automatic scope
- Start out life in stack memory
- Moved to heap during `Block_copy()`
- Beware: Address of `__block` variables may change!

Now what about this __block thing?

```
#include <stdio.h>
#include <Block.h>
int main(int argc, char *argv[]) {
  __block int i = 0;
  printf("&i = %p\n", &i);
  void (^b)() = ^{ ++i; };
  printf("&i = %p\n", &i);
  b = Block_copy(b);
  printf("&i = %p\n", &i);
  return 0;
}
```

```
$ ./a.out
&i = 0x7fff6a9a7a70
&i = 0x7fff6a9a7a70
&i = 0x10ae00868
```

Rule of thumb: Don't take address of __block variables!

Blocks in Objective-C

Blocks in Objective-C

- This is what makes blocks really useful: Blocks are Objective-C objects!
- Both block classes inherit `NSObject`
- There's `-copy` for `Block_copy()`
- and `-release` for `Block_release()`
- but there's also `-retain` ?!

Blocks are Objects

- The conventions for Objective-C objects say that `-retain` MUST return the same instance that it was called with. This means that retain cannot call `-copy`!
- This can lead to really nasty bugs!
- Rule of thumb: `-copy` and `-autorelease` blocks prior to storing them anywhere (properties, collections, etc.).

Blocks are Objects

```
typedef void (^Block)();

NSArray *f() { // wrong!
  return [NSArray arrayWithObject:^{ ... }];
}
```

```
NSArray *f() { // correct!
  return [NSArray arrayWithObject:[[^{
    ..
  } copy] autorelease]];
}

@property (retain) Block block; // bad idea!

@property (copy) Block block; // better safe than sorry!
```

Blocks are Objects

- Blocks in Objective-C have one more very important difference from blocks in C: <u>All local objects are automatically retained as they are referenced!</u>

```
- (void)someMethod
{
  id someObject = ...;
  ... ^{
    [someObject someMessage]; // retains someObject
  };
  ... ^{
    someIvar += 10; // retains self
  };
}
```

Blocks in Cocoa(Touch)

Blocks in Cocoa(Touch)

- Apple's use of blocks is currently limited
  - Only a few new APIs are using blocks to their full potential (i.e. AssetsLibrary)
  - Blocks support in core frameworks via 3rd party libraries

3rd Party Libraries

- BMKit (github.com/bmeurer/BMKit)
- BlockKit (github.com/nickpaulson/BlockKit)
- BlocksKit (github.com/zwaldowski/BlocksKit)
- etc.

Example from BMKit

```
typedef void (^BMBlock)();
@interface NSThread (BMKitAdditions)
- (id)initWithBlock:(BMBlock)aBlock;
@end

@implementation NSThread (BMKitAdditions)
+ (void)BM_invokeBlock:(BMBlock)aBlock {
  if (!aBlock) return;
  NSAutoreleasePool *pool = [NSAutoreleasePool new];
  aBlock();
  [pool drain];
}
- (id)initWithBlock:(BMBlock)aBlock {
  return [self initWithTarget:[NSThread class]
                     selector:@selector(BM_invokeBlock:)
                       object:[[aBlock copy] autorelease]];
}
@end
```

Grand Central Dispatch

Grand Central Dispatch

- Available for OS X 10.6+, iOS 4.0+, FreeBSD 8.1+
- “The key innovation of GCD is shifting the responsibility for managing threads and their execution from applications to the operating system” (Apple Marketing)
- But it’s more (Developer’s POV)

Grand Central Dispatch

- The Core: Task Parallelism based on Thread Pool Pattern
- Global thread pooling (Pthread Workqueues, XNU part)
- GCD based on threads, but hides (most) nasty details of concurrent programming
- Tightly integrated with Cocoa(Touch)

Grand Central Dispatch

- Works by queueing up tasks and scheduling them for execution depending on available processing resources (CPU only)
- Task either blocks or functions
- Work items can be associated with event sources (sockets, timers, etc.)
- Helps to avoid threading bugs (Deadlocks, Priority Inversion, etc.) by design

GCD building blocks

- Dispatch Queues
- Dispatch Groups
- Dispatch Sources
- Dispatch Semaphores

Dispatch Queues

- Maintain a queue of tasks and execute them on their turn
- Serial or concurrent
- Optimal scheduling based on availability
- Serial queues can avoid locks on shared resources
- Less code, easier to get right

Dispatch Groups

- Group several tasks
- Wait for completion of all grouped tasks
- Integrated with Dispatch Queues

Dispatch Semaphores

- Control concurrent execution of tasks
- Similar to POSIX Semaphores
- Better avoid them, use queues

Dispatch Sources

- Combine Dispatch Queues with event sources
- Several sources (sockets, timers, signals, etc.)
- Integration with CoreFoundation / Foundation run loops

Don’t block main thread

```
// Bad idea: blocks main thread / UI
- (IBAction)computeSomethingDidActivate:(id)sender {
   NSString *result = [_businessLogic computeSomething];
   _resultLabel.text = result;
}
```

```
// Do it asynchronously using dispatch queues!
- (IBAction)computeSomethingDidActivate:(id)sender {
   dispatch_async(dispatch_get_global_queue(
         DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
      NSString *result = [_businessLogic computeSomething];
      dispatch_async(dispatch_get_main_queue(), ^{
         _resultLabel.text = result;
      });
   });
}
```

Avoid locks

```
Resource *myResource = ...;
dispatch_queue_t myQueue =
  dispatch_queue_create(“com.example.myQueue”, NULL);
...
void doSomethingWithResource() {
  dispatch_async(myQueue, ^{
    // do whatever with myResource
    ...
  });
}
void doSomethingWithResourceAndWait() {
  dispatch_sync(myQueue, ^{
    // do whatever with myResource
    ...
  });
}
```

Parallelize loops

```
// A simple for loop
for (i = 0; i < count; ++i) {
  output[i] = process(input, i);
}

// Not as easy as OpenMP, but less trouble
dispatch_apply(count, dispatch_get_global_queue(
    DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^(size_t i) {
  output[i] = process(input, i);
});
```

GCD is not all magic

- APIs used concurrently must be thread-safe / reentrant!
- Deadlock / Priority Inversion less likely but still possible, i.e.

```
void deadlock(dispatch_queue_t queue) {
  dispatch_sync(queue, ^{
    dispatch_sync(queue, ^{
      // we never get here!
    });
  });
}
```

Real world example

- FreeBSD developer Robert Watson ported the Apache HTTP server to GCD
- Implemented as MPM (Multi-Processing Module)
- GCD MPM had 1/2 to 1/3 the number of lines as other thread MPMs

GCD advantages

- Multicore programming made easy
- No need to mess with threads, thread pools and locking issues
- Think in tasks and task queues
- Tightly integrated with Cocoa and Blocks
- You'll be addicted to GCD once you know it!

Summary

- Blocks puts back the fun in (Objective-)C programming (may also help C++)
- New functional style programming, a lot less code
- GCD brings multicore programming to the masses, no more threading headaches
- Available with OS X 10.6+ and iOS 4.0+ and recent clang on other platforms