

Blocks rewriting with clang

November 21, 2014, by Florent Bruneau

Introduction

Back in 2009, Snow Leopard was quite an exciting OS X release. It didn't focus on new user-visible features but instead introduced a handful of low level technologies. Two of those technologies **Grand Central Dispatch** (a.k.a. GCD) and **OpenCL** were designed to help developers benefit from the new computing power of modern computer architectures: multicore processors for the former and GPUs for the latter.

Alongside the GCD engine came the extension called **blocks**. Blocks are the C-based flavor of what is commonly known as the **closure**: a callable object that carries with it the context in which it was created. The syntax for blocks is very similar to the one used for functions, with the exception that the pointer star is * replaced by a caret ^. This allows inline definition of callbacks which often can help improving the readability of the code.

[1 Introduction](#)[2 Blocks are a desirable feature](#)[3 How it works](#)[4 Why we needed a rewriter](#)[5 What we did](#)[6 Limitations](#)[7 4 years later...](#)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 static void call_blk(void (^blk)(const char *str))
5 {
6     blk("world");
7 }
8
9 int main(int argc, char *argv[])
10 {
11     int count = argc > 1 ? atoi(argv[1]) : 1;
12
13     call_blk(^{const char *str} {
14         printf("Hello %s from %s!\n", str, argv[0]);
15     });
16     call_blk(^{const char *str} {
17         for (int i = 0; i < count; i++) {
18             printf("%s!\n", str);
19         }
20     });
21     return 0;
22 }
```

Blocks are a desirable feature

Standard C does not contain closures. **GCC supports nested functions** that are closures to some extent. However, nested functions cannot escape their definition scope, and therefore cannot be used in asynchronous code.

As a consequence, continuations in C are often implemented as a pair of a callback function and a context. The context contains variables needed by the callback to continue the execution of the program. Thus the type of the context is specific to the callback. That's why APIs that use continuations usually take a pair of a function and a void * pointer, which itself is given back to the function when it get called. This is very flexible, but deeply unsafe since using void * forbids any type checking by the compiler, the code can easily be broken during a refactoring, allowing a mismatch between the data expected by the callback and the actual content of the context it receives.

For example, this is how we could implement the previous example using callbacks instead of blocks:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 static void call_cb(void (*cb)(const char *str, void *ctx), void *ctx) {
5     (*cb)("world", ctx);
6 }
7
8
9 static void say_hello_from(const char *str, void *ctx)
10 {
11     const char *from = ctx;
12     printf("Hello %s from %s!\n", str, from);
13 }
14
15 static void repeat(const char *str, void *ctx)
16 {
17     int count = (int)(intptr_t)ctx;
18     for (int i = 0; i < count; i++) {
19         printf("%s!\n", str);
20     }
21 }
22
23
24 int main(int argc, char *argv[])
25 {
26     int count = argc > 1 ? atoi(argv[1]) : 1;
27
28 }
```

```
29 |     call_cb(&say_hello_from, argv[0]);
30 |     call_cb(&repeat, (void *)(&count));
31 |     return 0;
32 | }
```

On the other hand, blocks are type-safe. The compiler checks that the programmer is passing a block of the correct type and then identifies the variables from the parent scopes that are used by the block (we say those variables are *captured* by the block). It then automatically generates the code that creates and forwards the context. This ensures the context is always correct.

Blocks provide a safer and more concise syntax for a very common pattern.

How it works

In practice, the compiler does almost the same thing as we did with callback and context. It goes a bit further, though, by putting the pointer to the function in the context, that way a block is a single object that contains a pointer to the function and the associated data. Here is a second take on our rewrite of our example in plain (blockless) C. This second approach uses something very similar to what the compiler does with blocks:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct take_str_block {
5      void (*cb)(void *ctx, const char *str);
6  };
7
8  static void call_cb(const struct take_str_block *blk) {
9  {
10     (*blk->cb)(blk, "world");
11 }
12
13 struct say_hello_from_block {
14     void (*cb)(void *ctx, const char *str);
15     /* Captured variables */
16     const char *from;
17 };
18
19 static void say_hello_from(void *ctx, const char *str)
20 {
21     const struct say_hello_from_block *blk = ctx;
22
23     printf("Hello %s from %s!\n", str, from);
24 }
25
26 struct repeat_block {
27     void (*cb)(void *ctx, const char *str);
28     /* Captured variables */
29     int count;
30 };
31
32 static void repeat(void *ctx, const char *str)
33 {
34     const struct repeat_block *blk = ctx;
35
36     for (int i = 0; i < blk->count; i++) {
37         printf("%s!\n", str);
38     }
39 }
40
41 int main(int argc, char *argv[])
42 {
43     int count = argc > 1 ? atoi(argv[1]) : 1;
44
45     {
46         struct say_hello_from_block ctx = {
47             .cb = &say_hello_from,
48             .from = argv[0]
49         };
50         call_cb((struct take_str_block *)&ctx);
51     }
52     {
53         struct repeat_block ctx = {
54             .cb = &repeat,
55             .count = count
56         };
57         call_cb((struct take_str_block *)&ctx);
58     }
59     return 0;
60 }
```

Here, the block objects are structures that extend the block type expected by the function `call_cb`. This ensures the function receives an object it can manage and that each block implementation can add its own variables in its context. As you can see in this example, both `say_hello_from` and `repeat` extend the `take_str_block` type with their captured variables. With that approach, a cast is needed to downgrade the extended structure to the base type, but there is no required cast for captured variables anymore (no more `(intptr_t)` cast to pass an `int` into a `void *`).

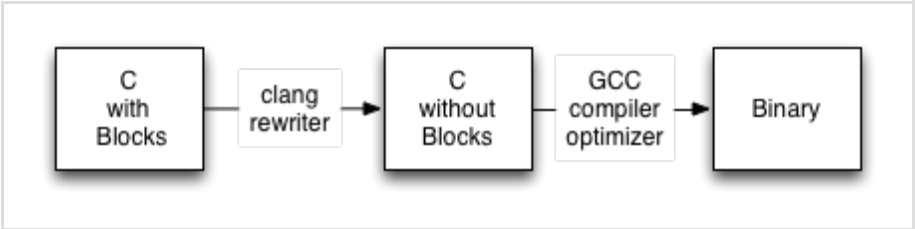
The actual ABI for blocks is a bit more complex since it must deal with captured variable shared modifications, block persistency, ... but the base principles are there.

Why we needed a rewriter

So, we wanted blocks in our code base. Back in 2010, clang was the only compiler supporting blocks within the Linux environment¹. However, at that time, clang was still a very young project and it just didn't fit our needs in terms of optimizations compared to GCC. On top of that, our software products ran (and have always been running) on RedHat Enterprise Linux, so using a compiler supported by RHEL was highly desirable. Unfortunately, this was not the case of clang, not even in its most recent releases.

Thankfully, clang contained a rewriting infrastructure that let a custom pass manipulate the AST². Even more interestingly, clang came with a built-in Objective-C to C++ rewriter... and since Objective-C includes blocks and C++ does not, that rewriter already did the work of rewriting blocks to C++.

However, our code is written in plain C (well, GNU C99 in fact), and we actually use the C99 features. If C++ was a strict superset of C, this wouldn't be an issue to rewrite our code to C++. Sadly, C++ is not strictly *backward* compatible with C and some syntax, like the **designated initializer** syntax I used in the previous code sample are **unavailable in C++**. As a consequence, rewriting our code to C++ is not acceptable as it would forbid the use of handy C99 features. This was (and still is) a no-brainer: we needed a rewriter from C with blocks to C supported by GCC, that way, we could both have blocks in C and use GCC to compile the code:



Thanks to the existing rewriter, this was not too hard³. Before starting to detail our process, let see what was the actual output of the origin Objective-C to C++ rewriter. The following code is what we obtained by running `clang -cc1 -rewrite-objc` on the program of the introduction of the article (for the sake of clarity, this excludes the large generated preamble):

```
1 struct __block_impl {
2     void *isa;
3     int Flags;
4     int Reserved;
5     void *FuncPtr;
6 };
7
8 #include <stdio.h>
9 #include <stdlib.h>
10
11 static void call_blk(void (*blk)(const char *str))
12 {
13     ((void (*)(struct __block_impl *, const char *))((struct __block_impl *)blk)->FuncPtr)((struct __block_impl *)blk,
14     "world");
15 }
16
17 struct __main_block_impl_0 {
18     struct __block_impl impl;
19     struct __main_block_desc_0* Desc;
20     char **argv;
21     __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, char **_argv, int flags=0) : argv(_argv) {
22         impl.isa = &__NSConcreteStackBlock;
23         impl.Flags = flags;
24         impl.FuncPtr = fp;
25         Desc = desc;
26     }
27 };
28 static void __main_block_func_0(struct __main_block_impl_0 *__cself, const char *str) {
29     char **argv = __cself->argv; // bound by copy
30
31     printf("Hello %s from %s!\n", str, argv[0]);
32 }
33
34 static struct __main_block_desc_0 {
35     size_t reserved;
36     size_t Block_size;
37 } __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0)};
38
39 struct __main_block_impl_1 {
40     struct __block_impl impl;
41     struct __main_block_desc_1* Desc;
42     int count;
43     __main_block_impl_1(void *fp, struct __main_block_desc_1 *desc, int _count, int flags=0) : count(_count) {
44         impl.isa = &__NSConcreteStackBlock;
45         impl.Flags = flags;
46         impl.FuncPtr = fp;
47         Desc = desc;
48     }
49 };
50 static void __main_block_func_1(struct __main_block_impl_1 *__cself, const char *str) {
51     int count = __cself->count; // bound by copy
52
53     for (int i = 0; i < count; i++) {
54         printf("%s!\n", str);
55     }
56 }
57
58 static struct __main_block_desc_1 {
```

```

59|   size_t reserved;
60|   size_t Block_size;
61| } __main_block_desc_1_DATA = { 0, sizeof(struct __main_block_impl_1)};
62| int main(int argc, char *argv[])
63| {
64|     int count = argc > 1 ? atoi(argv[1]) : 1;
65|
66|     call_blk((void (*)(const char *))&__main_block_impl_0((void *)__main_block_func_0, &__main_block_desc_0_DATA, argv));
67|     call_blk((void (*)(const char *))&__main_block_impl_1((void *)__main_block_func_1, &__main_block_desc_1_DATA, count));
68|     return 0;
69| }

```

If you look at this closely, you can notice the similarity with our `take_str_block` example. The compiler generates a function and an `impl` structure for each block. The base structure `__block_impl` contains the function pointer as well as information about the type of block (mostly used for block persistency).

What we did

First, we had to get rid of the Objective-C specific code: no need to keep code we don't care about. We initially hacked directly the code of the Objective-C to C++ rewriter. But it rapidly proved really hard to maintain the patched rewriter since we had a lot of conflicts with each new `clang` release, so we actually forked the Objective-C to C++ rewriter into a specialized block rewriter, introducing a `-rewrite-blocks` flag to `clang`.

Secondly, we needed to produce pure C code instead of C++ for blocks. As you can see in the generated code, the generated code uses structure constructors to initialize the implementation objects. Constructors do not exist in C, but we have initializers (and far better initializers than C++ ones). As a consequence, we changed the code that instantiated the `impl` structure to use structure initializers. Note that our implementation uses two C99 features: the designated initializers we mentioned already as well as **compound literals**.

The rewriter also uses another feature that cannot be observed in our example: the **RAII**, which is used only for variables with the `__block` modifier. Those variables are captured by reference instead of being captured by value. As a consequence, the runtime must perform some reference counting on those variables. The RAII is used to ensure we release the reference taken by the definition scope of the variable when we exit that scope. Standard C contains no equivalent construct. As a consequence, we had to manually generate the release of the reference everywhere we exited the definition scope of the variable. Thankfully, **GCC has a cleanup attribute**. This attribute associates a cleanup callback to a variable and acts exactly the same way as the RAII does. As a consequence, this completely solves our issue in a simple way.

With those issues fixed, here is the output of our rewriter `clang -cc1 -rewrite-blocks`:

```

1| struct __block_impl {
2|     void *isa;
3|     int Flags;
4|     int Reserved;
5|     void *FuncPtr;
6| };
7|
8| #include <stdio.h>
9| #include <stdlib.h>
10|
11| static void call_blk(void (*blk)(const char *str))
12| {
13|     ((void (*)(struct __block_impl *, const char *))((struct __block_impl *)blk)->FuncPtr)((struct __block_impl *)blk,
14| "world");
15| }
16|
17| struct __main_block_impl_0 {
18|     struct __block_impl impl;
19|     struct __main_block_desc_0* Desc;
20|     char **argv;
21| #define __main_block_impl_0(__blk_fp, __blk_desc, __argv, __blk_flags) \
22|     { \
23|         .argv = (__argv), \
24|         .impl = { \
25|             .isa = &__NSConcreteStackBlock, \
26|             .Flags = (__blk_flags), \
27|             .FuncPtr = (__blk_fp), \
28|         }, \
29|         .Desc = (__blk_desc), \
30|     }
31| #define __main_block_impl_0__INST(...) ({ \
32|     memcpy(&__main_block_impl_0__VAR, &(struct __main_block_impl_0)__main_block_impl_0(__VA_ARGS__), sizeof(__main_block_impl_0__VAR); \
33|     &__main_block_impl_0__VAR; \
34| })
35| };
36| static void __main_block_func_0(struct __main_block_impl_0 *__cself, const char *str) {
37|     char **argv = __cself->argv; // bound by copy
38|
39|     printf("Hello %s from %s!\n", str, argv[0]);
40| }
41|
42| static struct __main_block_desc_0 {
43|     unsigned long reserved;
44|     unsigned long Block_size;
45| } __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0)};
46|
47| struct __main_block_impl_1 {
48|     struct __block_impl impl;
49|     struct __main_block_desc_1* Desc;
50|     int count;
51| #define __main_block_impl_1(__blk_fp, __blk_desc, __count, __blk_flags) \

```

```
52 | { \
53 |     .count = (_count), \
54 |     .impl = { \
55 |         .isa = &_NSConcreteStackBlock, \
56 |         .Flags = (__blk_flags), \
57 |         .FuncPtr = (__blk_fp), \
58 |     }, \
59 |     .Desc = (__blk_desc), \
60 | }
61 | #define __main_block_impl_1__INST(...) ({ \
62 |     memcpy(&__main_block_impl_1__VAR, &(struct __main_block_impl_1)__main_block_impl_1(__VA_ARGS__), sizeof(__main_block_impl_1__
63 |     &__main_block_impl_1__VAR; \
64 | })
65 | };
66 | static void __main_block_func_1(struct __main_block_impl_1 *__cself, const char *str) {
67 |     int count = __cself->count; // bound by copy
68 |
69 |     for (int i = 0; i < count; i++) {
70 |         printf("%s!\n", str);
71 |     }
72 | }
73 |
74 | static struct __main_block_desc_1 {
75 |     unsigned long reserved;
76 |     unsigned long Block_size;
77 | } __main_block_desc_1_DATA = { 0, sizeof(struct __main_block_impl_1)};
78 | int main(int argc, char *argv[])
79 | {struct __main_block_impl_1 __main_block_impl_1__VAR;struct __main_block_impl_0 __main_block_impl_0__VAR;
80 |     int count = argc > 1 ? atoi(argv[1]) : 1;
81 |
82 |     call_blk((void (*)(const char *))(void *)__main_block_impl_0__INST((void *)__main_block_func_0, &__main_block_desc_0_DATA, a
83 | ));
84 |     call_blk((void (*)(const char *))(void *)__main_block_impl_1__INST((void *)__main_block_func_1, &__main_block_desc_1_DATA, c
85 | ));
86 |     return 0;
87 | }
```

Limitations

An issue remains, though, the rewriter cannot rewrite blocks within macros. This is a direct consequence of how the preprocessor works: it is a (not so) simple text replacement and the text of a macro is not guaranteed to be valid C. This could be worked around by rewriting the code output of the preprocessor instead of rewriting the source code. However, since we need to compile the output of the rewriter using GCC in order to use the optimizer (which is the most complex part of the compiler) supported by RHEL this was not possible. Indeed, in our code base, we sometimes use some compiler specific code paths (for example, until recently, **clang didn't know about the flatten attribute**, GCC didn't know about clang's `__has_feature()` preprocessor intrinsic...). Since the rewriter is based on clang, the macros are expanded to their clang flavour. As a consequence, had we worked on the preprocessed code, at best we would have lost some GCC-specific optimisations and at worst we would have encountered code GCC could not compile. Today, rewriting within macros remains the main limitation of our toolchain. In early versions, it even used to cause crashes of clang, a consequence of a pointer misuse inherited from the Objective-C rewriter⁴.

Since we rewrite the original source file, not the preprocessed file, we also cannot rewrite blocks in included files. This means that no blocks are allowed in header files (or in any other included code file). However, we still need to declare block types and functions that take blocks as arguments in those header files. And here comes probably the part that created the greatest amount of confusion among our engineers.

A block type is defined using a caret:

```
1 | typedef int (^my_block_b)(const char *str);
2 |
3 | int call_blk(int (^blk)(const char *str));
```

That won't compile with GCC. So we had to find a way to allow declaration of block types even in header files with two constraints: we must use the block syntax when the compiler supports blocks, but use some standard C syntax when the compiler does not. We did this by introducing a dedicated `BLOCK_CARET` macro that gets rewritten to a caret when the compiler supports blocks, but as a star when the compiler does not support them:

```
1 | #ifdef __BLOCKS__
2 | # define BLOCK_CARET  ^
3 | #else
4 | # define BLOCK_CARET  *
5 | #endif
6 |
7 | typedef int (BLOCK_CARET my_block_b)(const char *str);
8 |
9 | int call_blk(int (BLOCK_CARET blk)(const char *str));
```

This means that GCC will see block types as functions types. However we already saw that a block is not a function, but a structure. The rewriting to a function type is just a trick that makes the code acceptable to GCC, but that trick does not transform blocks to functions. Even if both blocks and functions are callable C types, they are not interchangeable: a function cannot be used where a block is expected, and vice-versa. Blocks remain a patch on top of the C standard. Most languages that have built-in closures won't have that kind of limitations, Apple itself fixed this in the Swift language... five years later.

4 years later...

We've been using our rewriter for over 4 years. During the few first months, we had a hard time convincing our engineers that blocks really were worth the extra cost (the extra compilation time and the various limitations of our rewriter), but today, a quarter of our code base uses blocks. The benefits of the lighter syntax

introduced by blocks can be observed in various use cases spanning from asynchronous code to the core of our database engines. For example, we can easily create atomic commits using blocks:

```
1 db_do_atomically(^{
2     do_some_write();
3     do_some_other_write();
4 });
```

We also spotted more limitations (mostly with `__block` variables), added basic support for C++ (ironically, we now think that we can generate better C++ than the original Objective-C to C++ rewriter for the small subset of C++ we support)... Probably the most frustrating issue today is the performance impact of blocks: they are too tricky to be inlined by GCC, even if defined and called in the same compilation unit. Moreover, in some use cases, we spotted limitation due to the fact blocks get allocated on the heap when we need to persist them (using `Block_copy()`), causing some contention in the allocator.

Our hacked clang is [available on github](#) for all branches since clang 3.0 (we actually began working on it before clang 3.0, but that early work was lost when we switched from our own git-svn clone to the official llvm git mirror). The code is not clean and someday we'll do a great cleanup pass in order to be able to submit our patches upstream.

-
1. though Apple's GCC fork supported it too ↩
 2. the AST, Abstract Syntax Tree, is a representation of the source code in the form of a tree which is much easier to manipulate than raw text ↩
 3. you can still find our original discussion about this in the [clang mailing list](#) ↩
 4. It looks like the Objective-C rewriter was some kind of experiment, and it's clearly not of production quality ↩

bookmark the **permalink** [<https://techtalk.intersec.com/2014/11/blocks-rewriting-with-clang/>] .



About Florent Bruneau

"When you do things right, people won't be sure you've done anything at all." - Futurama, 3x20 Godfellas
On twitter [@FlorentBruneau](#).

2 Comments


Intersec Tech Talk

1 Login

Recommend

Share





Sort by Best




Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?





Marty • 3 years ago


Hi, I've been following along at home.

I couldn't get the 2nd plain-C version to compile. My version of the main() function is below - do I have the right end of the stick?

```
int main( int argc, char *argv[] )
{
    int count = argc > 1 ? atoi(argv[1]) : 1;
    printf("Version 2\n");
    {
        struct say_hello_from_block ctx = {
            .cb = &say_hello_from,
            .from = argv[0]
        };
        call_cb( (struct take_str_block*)&ctx);
    }

    {
        struct repeat_block ctx = {
            .cb = &repeat,
            .count = count
        };
        call_cb( (struct take_str_block*)&ctx);
    }
    return 0;
}
```

^ | v • Reply • Share >



Florent Bruneau Mod ➔ Marty • 3 years ago

Indeed, we used the wrong variable. Thanks

^ | v • Reply • Share >

ALSO ON INTERSEC TECH TALK

More about locality

2 comments • 4 years ago •

Someone — Your function get_at has a security issue: If you pass negative values you can access memory outside of your array. Use size_t or at least unsigned integers for array ...

C Modules

1 comment • 2 years ago •

Pierre-Yves Caneill — Nice post :)

Memory – Part 6: Optimizing the FIFO and Stack allocators


3 comments • 3 years ago •


Nicholas Frechette — Indeed, if you need to reserve 2TB or more, it is far less practical. You must have quite the interesting scenario if you manage to push 1TB into a ...


One year at Intersec (actually two)

1 comment • 3 years ago •

Toni — Nice :-)And thanks for this post; this is the first that I can fully understand :-)

 Subscribe

 Add Disqus to your siteAdd DisqusAdd

 Privacy