How does it work - Meta macros

March 21, 2014 • 6 min read • original

21 Mar 2014

tl;dr

Meta macros are pretty nifty but trying to follow how they work can really challenge the limits of your mental stack frame.

Today I'm going to try and follow macro expansion from start to finish using libextobjc to do some basic metaprogramming. This is going to be a bumpy ride but as always it's great to see how different techniques can be used to solve problems.

Let's start with a fictitious problem that I would like to solve with some metaprogramming. I would probably never do this in a real project but it gives me a realistic use case to work through.

Imagine I want my view controllers to fail hard if I forget to connect up an outlet in the xib file. I could start with something like this:

```
 - (void)viewDidLoad;
{
  [super viewDidLoad];

  NSParameterAssert(self.firstNameTextField);
  NSParameterAssert(self.lastNameTextField);
  NSParameterAssert(self.passwordTextField);
  NSParameterAssert(self.passwordConfirmationTextField);
}
```

whoa that's a lot of repetition and it's not going to scale well. What would be great is if I could write some code that would write this repetitious code for me, ideally I would just type something like this:

```
 - (void)viewDidLoad;
{
  [super viewDidLoad];

  PASAssertConnections(firstNameTextField, lastNameTextField, passwordTextField,
passwordConfirmation);
}
```

This seems a lot DRY'er so let's aim for something similar to this and see how we get on.

`metamacro_foreach`

After examining the metamacros header I can see that there is a foreach macro that sounds like it would be perfect for this task.

The definition of `metamacro_foreach` looks like this:

```
#define metamacro_foreach(MACRO, SEP, ...)
```

After reading the docs I can see that the `MACRO` argument should be the name of a macro that takes two arguments in the form of `MACRO(INDEX, ARG)`. The `INDEX` parameter will be the index of the current iteration in the for loop and the `ARG` parameter will be the argument for the current iteration in the for loop.

So I need to start of by defining a macro that takes these two arguments and expands to the `NSParameterAssert` that I want. Here's a first stab at such a macro

```
#define OUTLET_ASSERT(INDEX, NAME) NSParameterAssert([self NAME])
```

I don't actually care to use the value of `INDEX` so it is ignored. This is the macro that will be used within the `metamacro_foreach` and will eventually expand into the required `NSParameterAssert` s.

In each of the following examples I'll show the input (starting macro) above the 3 dashes and what this would theoretically expand into below the 3 dashes. I'll optionally show any macro definitions at the top of the code block.

Here's how my `OUTLET_ASSERT` macro will work:

```
 OUTLET_ASSERT(0, firstNameTextField);

---

NSParameterAssert([self firstNameTextField]);
```

metamacro_foreach

Now let's see how we can use `metamacro_foreach` to write the `PASAssertConnections` macro that will take in a list of ivar names and expand them to the required `NSParameterAssert`s.

```
 #define metamacro_foreach(MACRO, SEP, ...) \
        metamacro_foreach_cxt(metamacro_foreach_iter, SEP, MACRO, __VA_ARGS__)

metamacro_foreach(OUTLET_ASSERT, ;, firstNameTextField, lastNameTextField)

---

metamacro_foreach_cxt(metamacro_foreach_iter, ;, OUTLET_ASSERT, firstNameTextField,
lastNameTextField)
```

In this case I pass `OUTLET_ASSERT` as the macro to use on each iteration. I pass `;` to use as a separator between iterations, which will terminate each `NSParameterAssert`. Then finally a comma separated list of ivar names that we are going to iterate over and generate the `NSParameterAssert`s for.

With the previous expansion there are now two new macros that we need to look up and understand `metamacro_foreach_cxt` and `metamacro_foreach_iter`. `metamacro_foreach_iter` is arguably the simpler of the two but it's not needed until the end so let's see how `metamacro_foreach_cxt` expands.

metamacro_foreach_cxt

```
 #define metamacro_foreach_cxt(MACRO, SEP, CONTEXT, ...) \
        metamacro_concat(metamacro_foreach_cxt, metamacro_argcount(__VA_ARGS__))
(MACRO, SEP, CONTEXT, __VA_ARGS__)

metamacro_foreach_cxt(metamacro_foreach_iter, ;, OUTLET_ASSERT, firstNameTextField,
lastNameTextField)

---

metamacro_concat(metamacro_foreach_cxt, metamacro_argcount(firstNameTextField,
lastNameTextField))(metamacro_foreach_iter, ;, OUTLET_ASSERT, firstNameTextField,
lastNameTextField)
```

Great when this macro expands it introduces 2 more macros to look up, `metamacro_concat` and `metamacro_argcount`.

`metamacro_concat` is the easier of the two so we'll take a look at that first.

metamacro_concat

```
 #define metamacro_concat(A, B) \
        metamacro_concat_(A, B)

#define metamacro_concat_(A, B) A ## B

metamacro_concat(metamacro_foreach_cxt, 2)

---

metamacro_foreach_cxt2
```

Cool so `metamacro_concat` just expands to `metamacro_concat_`, which then just joins the tokens together using `##`. So `metamacro_concat` just has the effect of joining it's two arguments into one string.

Now we need to jump back to see how `metamacro_argcount` works

metamacro_argcount

```
 #define metamacro_argcount(...) \
        metamacro_at(20, __VA_ARGS__, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9,
8, 7, 6, 5, 4, 3, 2, 1)

metamacro_argcount(firstNameTextField, lastNameTextField)

---

metamacro_at(20, firstNameTextField, lastNameTextField , 20, 19, 18, 17, 16, 15, 14,
13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
```

The `metamacro_argcount` macro uses another macro called `metamacro_at`. The `metamacro_at` is similar in concept to indexing into an array like `myArray[index]`. In plain English this macro is the same as "give me the `nth` item in the following list".

The `metamacro_argcount` macro uses a clever little trick. If we put the numbers from `INDEX` down to `0` into an array and then ask for the value at `INDEX` we would get the last number, which would be 0. If we preprend something to the beginning of this array and asked for the value at `INDEX` again we would now get `1`.

Let's see this in Objective-C so it's easier to picture:

```
NSInteger index = 3;

@[                             @3, @2, @1, @0 ][index]; //=> @0 - 0 args
@[ @"argument",                @3, @2, @1, @0 ][index]; //=> @1 - 1 arg preprended
@[ @"argument", @"argument", @3, @2, @1, @0 ][index]; //=> @2 - 2 args preprended
```

The relationship is that when you prepend an argument to the array you shift all of the numeric values to the right by one step, which moves a higher number into the index that is being fetched. This of course only works up to the value of `INDEX` - so we can tell that this particular implementation of metamacros only supports 20 arguments.

NB - this implementation of metamacros requires at least one argument to be given when using `metamacro_argcount`.

You'll see the trick of inserting `__VA_ARGS__` into argument lists at different points used a few times so it's worth making sure you understand what is happening above.

Ok so that makes sense but what about `metamacro_at`?

metamacro_at

```
#define metamacro_at(N, ...) \
        metamacro_concat(metamacro_at, N)(__VA_ARGS__)
```

Great there's our old friend `metamacro_concat` so we don't need to look up how that works again to know that this will expand like this:

```
metamacro_at(20, firstNameTextField, lastNameTextField , 20, 19, 18, 17, 16, 15, 14,
13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)

---

metamacro_at20(firstNameTextField, lastNameTextField , 20, 19, 18, 17, 16, 15, 14,
13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
```

The change is very subtle. The `20` has moved from being an argument to now actually being part of the macro name. So now we need to look up `metamacro_at20`

```
#define metamacro_at20(_0, _1, _2, _3, _4, _5, _6, _7, _8, _9, _10, _11, _12, _13,
_14, _15, _16, _17, _18, _19, ...) metamacro_head(__VA_ARGS__)
```

It turns out that there are variants of `metamacro_at` defined for `0` to `20`, which allows you to access any of the first 20 arguments from the `__VA_ARGS__` arguments.

This is another common trick you'll see with metamacros, at some point you have to knuckle down and write out multiple versions of the same macro to handle different length argument lists. You'll often see that metamacros are generated by other scripts that allow you to specify how many arguments you would like to support without having to hand roll all the variations of `metamacro_at0..N`.

To make `metamacro_at` a little easier to digest I'll examine one of the smaller versions of this macro.

```
#define metamacro_at2(_0, _1, ...) metamacro_head(__VA_ARGS__)

metamacro_at2(firstNameTextField, lastNameTextField, passwordTextField,
passwordConfirmationTextField)

---

metamacro_head(passwordTextField, passwordConfirmationTextField)
```

The `_0` and `_1` arguments are basically used as placeholders to gobble up the items at indices `0` and `1` from the arguments. Then we bundle the rest of the arguments together with `...`. The newly trimmed `__VA_ARGS__` is then passed into `metamacro_head`

metamacro_head

```
#define metamacro_head(...) \
        metamacro_head_(__VA_ARGS__, 0)

#define metamacro_head_(FIRST, ...) FIRST

metamacro_head(passwordTextField, passwordConfirmationTextField)

---

passwordTextField
```

`metamacro_head` uses the opposite trick to `metamacro_at*`. In this case we are only interested in the first item and we want to throw away the rest of the `__VA_ARGS__` list. This is achieved by grabbing the first argument in `FIRST` and then collecting the rest with `...`.

Wow that escalated quickly. We now need to unwind out mental stack frame back to `metamacro_foreach_cxt`.

`metamacro_foreach_cxt`

Now we are more enlightened we can go back and expand both `metamacro_concat` and `metamacro_argcount` in the following:

```
#define metamacro_foreach_cxt(MACRO, SEP, CONTEXT, ...) \
        metamacro_concat(metamacro_foreach_cxt, metamacro_argcount(__VA_ARGS__))
(MACRO, SEP, CONTEXT, __VA_ARGS__)

metamacro_concat(metamacro_foreach_cxt, metamacro_argcount(firstNameTextField,
lastNameTextField))(metamacro_foreach_iter, ;, OUTLET_ASSERT, firstNameTextField,
lastNameTextField)


---


metamacro_foreach_cxt2(metamacro_foreach_iter, ;, OUTLET_ASSERT, firstNameTextField,
lastNameTextField)
```

Don't worry the end is now very much in sight, just a couple more painless macro expansions. The previous expansion gives us the new `metamacro_foreach_cxt2` macro to check out.

`metamacro_foreach_cxt2`

This is another example of macro that has multiple versions defined from `0..20`. Each of these foreach macros works by utilising the foreach macro that is defined to take one less argument than itself until we get all the way down to `metamacro_foreach_cxt1`

```
#define metamacro_foreach_cxt2(MACRO, SEP, CONTEXT, _0, _1) \
    metamacro_foreach_cxt1(MACRO, SEP, CONTEXT, _0) \
    SEP \
    MACRO(1, CONTEXT, _1)

metamacro_foreach_cxt2(metamacro_foreach_iter, ;, OUTLET_ASSERT, firstNameTextField,
lastNameTextField)
```

We are now at the point where we need to see what `MACRO` expands to. In this case `MACRO` is actually the `metamacro_foreach_iter` macro that we passed in near the beginning and I delayed explaining.

`metamacro_foreach_iter`

This macro is really just an implementation detail and as such shouldn't be used directly but we still want to see what part it plays:

```
 #define metamacro_foreach_iter(INDEX, MACRO, ARG) MACRO(INDEX, ARG)

metamacro_foreach_iter(0, OUTLET_ASSERT, firstNameTextField)

---

OUTLET_ASSERT(0, firstNameTextField)
```

Nice and simple - `metamacro_foreach_iter` is just a helper that takes our macro `OUTLET_ASSERT` and the two arguments that our macro should receive and puts the pieces in the right order to be further expanded into the `NSParameterAssert` calls that we want.

Thankfully that was only a minor detour so let's get right back to `metamacro_foreach_cxt2`

```
 #define metamacro_foreach_cxt2(MACRO, SEP, CONTEXT, _0, _1) \
    metamacro_foreach_cxt1(MACRO, SEP, CONTEXT, _0) \
    SEP \
    MACRO(1, CONTEXT, _1)

metamacro_foreach_cxt2(metamacro_foreach_iter, ;, OUTLET_ASSERT, firstNameTextField,
lastNameTextField)

---

metamacro_foreach_cxt1(metamacro_foreach_iter, ;, OUTLET_ASSERT, firstNameTextField)
\
    ; \
    OUTLET_ASSERT(1, lastNameTextField)
```

If you have gotten this far then the above is nothing special so we can progress straight to the next step:

```
 #define metamacro_foreach_cxt1(MACRO, SEP, CONTEXT, _0) MACRO(0, CONTEXT, _0)

metamacro_foreach_cxt1(metamacro_foreach_iter, ;, OUTLET_ASSERT, firstNameTextField)
\
    ; \
    OUTLET_ASSERT(1, lastNameTextField)

---

    OUTLET_ASSERT(0, firstNameTextField) \
    ; \
    OUTLET_ASSERT(1, lastNameTextField)
```

And that's it - we've followed the `metamacro_foreach` macro from the beginning of it's use all the way to it's end expansion and hopefully our heads are still in one piece.

Wrapping up

At the beginning of the post I said I was aiming for

```
 PASAssertConnections(firstNameTextField, lastNameTextField, passwordTextField,
passwordConfirmation);
```

now I'm actually one step away from achieving this, but if this post has gotten your interest I'll leave that as a simple exercise - it's always better to learn by doing and not just skimming through blog posts hoping to learn by osmosis.

Metaprogramming is normally something that people associate with more dynamic languages like Ruby but there's a whole load of possibilities and cool tricks out there just waiting to be learned. As always I encourage you to join me in peeling back the curtain and seeing that there is normally no magic to be found in your favorite OSS projects.

Original URL:
http://paul-samuels.com/blog/2014/03/21/how-does-it-work-meta-macros/