Declaring and Creating Blocks

Declaring a Block Reference

Block variables hold references to blocks. You declare them using syntax similar to that you use to declare a pointer to a function, except that you use `^` instead of `*`. The block type fully interoperates with the rest of the C type system. The following are all valid block variable declarations:

```
void (^blockReturningVoidWithVoidArgument)(void);
int (^blockReturningIntWithIntAndCharArguments)(int, char);
void (^arrayOfTenBlocksReturningVoidWithIntArgument[10])(int);
```

Blocks also support variadic (`...`) arguments. A block that takes no arguments must specify `void` in the argument list.

Blocks are designed to be fully type safe by giving the compiler a full set of metadata to use to validate use of blocks, parameters passed to blocks, and assignment of the return value. You can cast a block reference to a pointer of arbitrary type and vice versa. You cannot, however, dereference a block reference via the pointer dereference operator (`*`)—thus a block's size cannot be computed at compile time.

You can also create types for blocks—doing so is generally considered to be best practice when you use a block with a given signature in multiple places:

```
typedef float (^MyBlockType)(float, float);

MyBlockType myFirstBlock = // ... ;
MyBlockType mySecondBlock = // ... ;
```

Creating a Block

You use the `^` operator to indicate the beginning of a block literal expression. It may be followed by an argument list contained within `()`. The body of the block is contained within `{}`. The following example defines a simple block and assigns it to a previously declared variable (`oneFrom`)—here the block is followed by the normal `;` that ends a C statement.

```
float (^oneFrom)(float);

oneFrom = ^(float aFloat) {
    float result = aFloat - 1.0;
    return result;
};
```

If you don't explicitly declare the return value of a block expression, it can be automatically inferred from the contents of the block. If the return type is inferred and the parameter list is `void`, then you can omit the `(void)` parameter list as well. If or when multiple return statements are present, they must exactly match (using casting if necessary).

Global Blocks

At a file level, you can use a block as a global literal:

```
#import <stdio.h>

int GlobalInt = 0;
int (^getGlobalInt)(void) = ^{ return GlobalInt; };
```