Language Specification for Blocks

Revisions

2008/2/25 — created
2008/7/28 — revised, __block syntax
2008/8/13 — revised, Block globals
2008/8/21 — revised, C++ elaboration
2008/11/1 — revised, __weak support
2009/1/12 — revised, explicit return types
2009/2/10 — revised, __block objects need retain

Overview

A new derived type is introduced to C and, by extension, Objective-C, C++, and Objective-C++

The Block Type

Like function types, the Block type is a pair consisting of a result value type and a list of parameter types very similar to a function type. Blocks are intended to be used much like functions with the key distinction being that in addition to executable code they also contain various variable bindings to automatic (stack) or managed (heap) memory.

The abstract declarator,

```
int (^)(char, float)
```

describes a reference to a Block that, when invoked, takes two parameters, the first of type char and the second of type float, and returns a value of type int. The Block referenced is of opaque data that may reside in automatic (stack) memory, global memory, or heap memory.

Block Variable Declarations

A variable with Block type is declared using function pointer style notation substituting ^ for *. The following are valid Block variable declarations:

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

Variadic ... arguments are supported. [variadic.c] A Block that takes no arguments must specify void in the argument list [voidarg.c]. An empty parameter list does not represent, as K&R provide, an unspecified argument list. Note: both gcc and clang support K&R style as a convenience.

A Block reference may be cast to a pointer of arbitrary type and vice versa. [cast.c] A Block reference may not be dereferenced via the pointer dereference operator *, and thus a Block's size may not be computed at compile time. [sizeof.c]

Block Literal Expressions

A Block literal expression produces a reference to a Block. It is introduced by the use of the ^ token as a unary operator.

```
Block_literal_expression ::=   ^ block_decl compound_statement_body
block_decl ::=
block_decl ::= parameter_list
block_decl ::= type_expression
```

where type expression is extended to allow ^ as a Block reference (pointer) where * is allowed as a function reference (pointer).

The following Block literal:

```
^ void (void) { printf("hello world\n"); }
```

produces a reference to a Block with no arguments with no return value.

The return type is optional and is inferred from the return statements. If the return statements return a value, they all must return a value of the same type. If there is no value returned the inferred type of the Block is void; otherwise it is the type of the return statement value.

If the return type is omitted and the argument list is (void), the (void) argument list may also be omitted.

So:

```
^ ( void ) { printf("hello world\n"); }
```

and:

```
^ { printf("hello world\n"); }
```

are exactly equivalent constructs for the same expression.

The type_expression extends C expression parsing to accommodate Block reference declarations as it accommodates function pointer declarations.

Given:

```
typedef int (*pointerToFunctionThatReturnsIntWithCharArg)(char);
pointerToFunctionThatReturnsIntWithCharArg functionPointer;
^ pointerToFunctionThatReturnsIntWithCharArg (float x) { return functionPointer; }
```

and:

```
^ int ((*)(float x))(char) { return functionPointer; }
```

are equivalent expressions, as is:

```
^(float x) { return functionPointer; }
```

[returnfunctionptr.c]

The compound statement body establishes a new lexical scope within that of its parent. Variables used within the scope of the compound statement are bound to the Block in the normal manner with the exception of those in automatic (stack) storage. Thus one may access functions and global variables as one would expect, as well as static local variables. [testme]

Local automatic (stack) variables referenced within the compound statement of a Block are imported and captured by the Block as const copies. The capture (binding) is performed at the time of the Block literal expression evaluation.

The compiler is not required to capture a variable if it can prove that no references to the variable will actually be evaluated. Programmers can force a variable to be captured by referencing it in a statement at the beginning of the Block, like so:

```
(void) foo;
```

This matters when capturing the variable has side-effects, as it can in Objective-C or C++.

The lifetime of variables declared in a Block is that of a function; each activation frame contains a new copy of variables declared within the local scope of the Block. Such variable declarations should be allowed anywhere [testme] rather than only when C99 parsing is requested, including for statements. [testme]

Block literal expressions may occur within Block literal expressions (nest) and all variables captured by any nested blocks are implicitly also captured in the scopes of their enclosing Blocks.

A Block literal expression may be used as the initialization value for Block variables at global or local static scope.

The Invoke Operator

Blocks are invoked using function call syntax with a list of expression parameters of types corresponding to the declaration and returning a result type also according to the declaration. Given:

```
int (^x)(char);
void (^z)(void);
int (^(*y))(char) = &x;
```

the following are all legal Block invocations:

```
x('a');
(*y)('a');
(true ? x : *y)('a')
```

The Copy and Release Operations

The compiler and runtime provide copy and release operations for Block references that create and, in matched use, release allocated storage for referenced Blocks.

The copy operation Block_copy() is styled as a function that takes an arbitrary Block reference and returns a Block reference of the same type. The release operation, Block_release(), is styled as a function that takes an arbitrary Block reference and, if dynamically matched to a Block copy operation, allows recovery of the referenced allocated memory.

The __block Storage Qualifier

In addition to the new Block type we also introduce a new storage qualifier, __block, for local variables. [testme: a __block declaration within a block literal] The __block storage qualifier is mutually exclusive to the existing local storage qualifiers auto, register, and static. [testme] Variables qualified by __block act as if they were in allocated storage and this storage is automatically recovered after last use of said variable. An implementation may choose an optimization where the storage is initially automatic and only “moved” to allocated (heap) storage upon a Block_copy of a referencing Block. Such variables may be mutated as normal variables are.

In the case where a __block variable is a Block one must assume that the __block variable resides in allocated storage and as such is assumed to reference a Block that is also in allocated storage (that it is the result of a Block_copy operation). Despite this there is no provision to do a Block_copy or a Block_release if an implementation provides initial automatic storage for Blocks. This is due to the inherent race condition of potentially several threads trying to update the shared variable and the need for synchronization around disposing of older values and copying new ones. Such synchronization is beyond the scope of this language specification.

Control Flow

The compound statement of a Block is treated much like a function body with respect to control flow in that goto, break, and continue do not escape the Block. Exceptions are treated *normally* in that when thrown they pop stack frames until a catch clause is found.

Objective-C Extensions

Objective-C extends the definition of a Block reference type to be that also of id. A variable or expression of Block type may be messaged or used as a parameter wherever an id may be. The converse is also true. Block references may thus appear as properties and are subject to the assign, retain, and copy attribute logic that is reserved for objects.

All Blocks are constructed to be Objective-C objects regardless of whether the Objective-C runtime is operational in the program or not. Blocks using automatic (stack) memory are objects and may be messaged, although they may not be assigned into __weak locations if garbage collection is enabled.

Within a Block literal expression within a method definition references to instance variables are also imported into the lexical scope of the compound statement. These variables are implicitly qualified as references from self, and so self is imported as a const copy. The net effect is that instance variables can be mutated.

The Block_copy operator retains all objects held in variables of automatic storage referenced within the Block expression (or form strong references if running under garbage collection). Object variables of __block storage type are assumed to hold normal pointers with no provision for retain and release messages.

Foundation defines (and supplies) -copy and -release methods for Blocks.

In the Objective-C and Objective-C++ languages, we allow the __weak specifier for __block variables of object type. If garbage collection is not enabled, this qualifier causes these variables to be kept without retain messages being sent. This knowingly leads to dangling pointers if the Block (or a copy) outlives the lifetime of this object.

In garbage collected environments, the __weak variable is set to nil when the object it references is collected, as long as the __block variable resides in the heap (either by default or via Block_copy()). The initial Apple implementation does in fact start __block variables on the stack and migrate them to the heap only as a result of a Block_copy() operation.

It is a runtime error to attempt to assign a reference to a stack-based Block into any storage marked __weak, including __weak __block variables.

C++ Extensions

Block literal expressions within functions are extended to allow const use of C++ objects, pointers, or references held in automatic storage.

As usual, within the block, references to captured variables become const-qualified, as if they were references to members of a const object. Note that this does not change the type of a variable of reference type.

For example, given a class Foo:

```
Foo foo;
Foo &fooRef = foo;
Foo *fooPtr = &foo;
```

A Block that referenced these variables would import the variables as const variations:

```
const Foo block_foo = foo;
Foo &block_fooRef = fooRef;
Foo *const block_fooPtr = fooPtr;
```

Captured variables are copied into the Block at the instant of evaluating the Block literal expression. They are also copied when calling Block_copy() on a Block allocated on the stack. In both cases, they are copied as if the variable were const-qualified, and it’s an error if there’s no such constructor.

Captured variables in Blocks on the stack are destroyed when control leaves the compound statement that contains the Block literal expression. Captured variables in Blocks on the heap are destroyed when the reference count of the Block drops to zero.

Variables declared as residing in __block storage may be initially allocated in the heap or may first appear on the stack and be copied to the heap as a result of a Block_copy() operation. When copied from the stack, __block variables are copied using their normal qualification (i.e. without adding const). In C++11, __block variables are copied as x-values if that is possible, then as l-values if not; if both fail, it’s an error. The destructor for any initial stack-based version is called at the variable’s normal end of scope.

References to this, as well as references to non-static members of any enclosing class, are evaluated by capturing this just like a normal variable of C pointer type.

Member variables that are Blocks may not be overloaded by the types of their arguments.