WG14/N1370

Blaine Garst
Apple, Inc.
March 10, 2009

APPLE'S EXTENSIONS TO C

Background

Apple publishes and invites contributions to the source for its Mac OS X operating system and uses it for its Macintosh®, iPhone™, and other products. System level code is written primarily in C and C++ whereas Graphical User Interface code is primarily written in Objective-C. The open source *gcc* compiler is used to build these products internally and is provided externally as part of the Xcode® Developer Tools package. Apple is investing heavily in the open source LLVM compiler technology as an alternative to *gcc*.

In recent years Apple invested significantly in extending Objective-C by adding, among other things, optional garbage collection. This effort required extending C itself although Apple does not actively promote this usage. Most recently Apple has added a form of closures (called Blocks) to C, Objective-C, and C++. Although first class closures have been somewhat exclusively provided in garbage collected environments, Apple has chosen an implementation that works equally well in collected and uncollected environments.

This paper presents the C language changes required for these efforts.

Garbage Collection

Apple uses runtime helper functions for assignments to pointers that reference collectable memory. When actually running garbage collection these perform what is known as "write-barriers" in order to detect and track stores of pointers to GC heap nodes into other nodes and global variables. These "write-barriers" enable not only an efficient incremental ("generational") collections but even more importantly extremely efficient thread local objects, caches, and collections. GC nodes can be allocated, used, and recovered without any synchronization as long as they don't escape the stack. In general terms this is usually around half of all nodes allocated and scales well on multi-cpu and multi-core systems.

Although generational the Apple collector is a conservative non-copying collector (node addresses are never updated). Allocation can proceed during collections and at no point are all threads stopped. As such the collector is concurrent with the proviso that threads can block during a GC critical section for a duration generally measured in microseconds. We believe that this meets the pragmatic needs of many if not most programmers.

In section 6.2.4 the C standard speaks of three storage durations for objects: *auto* (and *register*) for stack based variables, *static* for globals and local statics, and *allocated*. *allocated* storage is acquired and released using malloc() and free(). The *const* type qualifier restricts mutation by forbidding modifications to objects whose type is so qualified. *volatile* imposes restrictions on optimizations.

In Apple's C we introduce two new type qualifiers, __*strong* and __*weak*, for use on pointer types. Assignments to these objects require the use of runtime support functions that do appropriate synchronization and notification with the collector. __*weak* is further restricted for use only on pointer objects of global storage and also introduces the requirement of a read access support function that implements a read-barrier.

__strong pointers hold references to garbage collected nodes. If referenced from the stack or global memory either directly or transitively, or are otherwise marked for preservation, such nodes are deemed reachable and are considered alive and survive collections. Unreachable nodes are deemed garbage and are collected and the storage handed back to their allocator. The collector can be configured to run on its own thread on memory demand or strictly on active request.

__weak pointers also hold references to garbage collected nodes but are ignored during reachability analysis. __weak pointer variables found to be referencing garbage are set to the null pointer value (e.g. cleared, a.k.a. zeroing weak) by the collector as part of the collection process.

Allocation of collectable nodes is done through new functions and not through malloc(). As an example, one allocates unscanned storage, that is, storage that does not reference other gc nodes, using a function similar to this:

```
char *__strong string =
         (char *__strong)malloc_unscanned(strlen(buffer)+1);
```

Since the collector does not track interior pointers we define the type of string+1 to be char * and so ++string is a type assignment error.

An array of pointers to such strings would be declared and allocated:

```
char *__strong *__strong array =
           (char *__strong *__strong)malloc_scanned(n*sizeof(char *));
```

Whereas `*string = 'a'` is a simple assignment of a character, `*array = string` requires a helper assignment function.

Since garbage collection requires the cooperation of the programmer by imposing restrictions on pointer manipulations it cannot be regarded as a fully automatic memory management system. Additional functions such as a `memmove_scanned()` must be defined and provided for convenient and correct programming in a collected environment. It is possible and even advisable to provide an explicit `free_collectable()` for those cases where memory efficiency is of paramount concern.

Issues

There are a number of interesting issues raised by the need to keep a GC pointer live and unaltered in its representation both during and beyond its last apparent use. In practice dead variable elimination has posed a small problem while derived interior pointers were still in use has been a small problem. It may be that dead __strong pointer variables should never be eliminated.

Storing the contents of a __strong variable into a non-__strong variable is both necessary and safe and yet violates the existing type qualifier constraints.

```
char *__strong gc_string = ...;
char * string_walker = gc_string + 0; // safe
char * wannabe_walker = gc_strong;  // type violation
```

Blocks

A closure is a function expression that closes over and preserves its lexical scope. A closure that is a first class object can be stored and any variables referenced from that lexical scope must be preserved beyond the lifetime of the declaring activation frame.

Apple has introduced a new compound type called a Block reference much like but distinct from a function pointer, a Block literal expression value syntax, and a new storage duration class for variables referenced from closures that live beyond the declaring activation frame.

Block references

A Block reference is declared like a function pointer but using the ^ token instead of *.

```
int (^blockReturningIntTakingCharArg)(char);
```

The usual composition rules apply for arrays of Block references or Blocks that return function pointers etc.

One invokes a Block reference by reusing function call syntax with appropriate arguments:

```
int a = blockReturningIntTakingCharArg('a');
```

Block literals

A Block literal is creating by introducing the use of the ^ token as a unary Block construction operator:

```
blockReturningIntTakingChar = ^int (char arg) { return arg*10; };
```

This creates and assigns a reference to an opaque Block data structure held in *auto* (stack) storage. The lifetime of the Block is that of its declaration "block" and as such the reference is somewhat precarious. For many purposes Block literals are directly passed to functions taking Block arguments:

```
qsort_b(array, nItems, size, ^int (void *item1, void *item2) { ... });
```

The power of a closure is its ability to reference local variables and act upon them while executing in a different activation frame. C API often handles this by allowing a context pointer to be passed in and passed along to a corresponding function and having the custom function treat the opaque context value as a custom structure with appropriate parameterization (see qsort_r). This is cumbersome.

In a callback case, say with HTML or XML parsers, or network triggers, there are tedious issues of when the context pointer is last used so that any allocated memory used for it can be recovered.

For callback purposes Apple provides Block_copy and Block_release operators that operate upon a single parenthesized Block reference expression (similar to *sizeof*). A Block_copy operation preserves the type of its argument. A Block_release operation yields a void value. When paired they recover any associated allocated memory.

Within the body of a Block literal one can declare local variables, one can reference global or static local variables, and one can call global functions. References to variables of auto storage are treated specially, however. Rather than secretly hoist such variables into constructs that might need to be preserved beyond the stack activation frame, the Apple specification and implementation decrees that references to variables of auto storage are imported at the time of Block literal construction as const copies.

```
int a = 10;
void (^block)(void) = ^void (void) { ++a; }; // forbidden!
```

The dominant use of variables in lexical scope is non-mutating. When a Block is copied the already captured value in the opaque stack based Block structure is bit copied to the heap version. (Subsequent Block_copy operations on a heap based Block merely update a reference counter).

The Block literal notation can be cumbersome and as such Apple provides two abbreviated forms. First, the return type of a Block literal can be omitted and it will be inferred from the type of any return statement value. Second, if the return type is omitted and the argument list is exactly (void) then the argument list can also be omitted. Thus:

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

and

```
^{ printf(“hello world\n”); }
```

are exactly equivalent.

__block variables

For the occasional yet important case where local variables are desired to be updated, as is traditional in prior closure implementations, Apple introduces a new storage-class-specifier known as and declared by the *__block* keyword.

A now common pattern in Apple code is this:

```
__block int max = 0;
IterateOverTreeOfInts(theIntTree, options, ^void (int member) {
         if (member > max) max = member;
});
```

where a __block variable is set to nil and some form of search or iteration code is used to find and set the variable. Multiple __block variables can of course be set.

Synchronous uses of Blocks are quite common so Apple has chosen to implement __block variables on the stack just as they have done for the initial Block literal. It is cheap and efficient.

In the more rare case where a Block is copied then referenced __block variables are also copied to the heap (if not already there) and are preserved for the duration of the activation frame or the last referencing heap based Block.

Although not yet specified the & operator should likely be forbidden upon __block storage as it is with register.

Uses

To better utilize multi-core processors Apple is introducing new APIs known as Grand Central Dispatch that schedule and perform Blocks on available cores with a variety of coordination options. In addition to `qsort_b()` Apple is providing iterator APIs that take Blocks as a more efficient and convenient way of processing common collection data structures. Cumbersome callback APIs are being augmented by much simpler Block APIs that let the programmer focus on what she needs to do rather than the tedious bookkeeping required to set up and recover custom callback data.

Blocks have been extended in C++ to allow stack objects to be const copied in Blocks or alternatively to be valid __block object types. Objective-C extend Blocks in several ways, the primary one being the ability to treat a Block regardless of origin in any form of code as an Objective-C object and in particular to participate in garbage collection.

Issues

The choice of starting Blocks on the stack as opposed to the heap was done primarily because the creation of a Block literal is done with a language construct but its recovery must be done, absent garbage collection, by explicit code. Although it is desirable to write:

```
return ^int (int x) { return x*10; };
```

it is our experience that creating and transferring ownership responsibility across APIs is very error prone in uncollected environments. Short of garbage collection it is possible to provide a reference-counted system where per-thread pools accumulate temporary objects such as what would be generated above, but this requires cooperation with either the threading system or all of its clients. It is in fact a form of garbage collection but with very little that is automatic about it.

There is also the cost of malloc to consider and whether it is cheap enough for the myriad of synchronous uses for which it is unnecessary.

These are, however, somewhat implementation specific choices.

Conclusions

Garbage Collection and Blocks are powerful well-known techniques to improve programmer productivity with very modest runtime costs and penalties. With two open source compiler implementations and open source runtime implementations they can be considered for use beyond the Apple platform.

References

All things LLVM: http://llvm.org/, including the C Language parser: http://clang.llvm.org/

Blocks specification http://clang.llvm.org/docs/BlockLanguageSpec.txt

Blocks ABI for Apple: http://clang.llvm.org/docs/BlockImplementation.txt