N2030: A Closure for C

Blaine Garst
2016/03/11-17
version 2

1 Introduction

This paper proposes the consideration of a *closure* construct for C2x. The key idea of a closure is that of a function expression with lexical access to variables declared in the enclosing scope whose lifetime can exceed the lifetime of that enclosing scope. This allows a convenient and efficient syntactic expression of a *unit of work*, a *task* in ISO/IEC parlance.

1.1 History

Closures are to be found in Alonso Church's original paper on lambda. Early implementations of LISP, however, used dynamic scoping, and closures were introduced by Guy Steele in SCHEME and then into more modern versions of LISP. The Smalltalk language uses closures as the implementation of compound statement in constructs such as if-then-else and loops. Ruby provides a simple form of closure and uses it extensively for collection iteration and file (stream) processing.

1.2 Background

This proposal has its genesis within Apple, Inc. in late December 2007 and quickly progressed to an alpha ship in May 2008, syntax changes and a full implementation followed by about August 2008, and a final release as part of Apple's 10.6 "Snow Leopard" release in August 2009. This release was notable at the time for two things: it was free, and it had no features other than performance. The performance gain was quite substantial and the result of extensive rework to eliminate thread pools in favor of a library approach to concurrency based on closures as the *unit of work*.

Closures, at Apple, were introduced simultaneously to the C language as well as Objective-C and C++. A concurrency library, with APIs in C, uses the closure form of `void (*)(void)` to capture anonymous work with no parameters and no return value. This type is available to all three languages, each of which had substantial communities within Apple. For C++, closures support capturing any object with a copy constructor, for Objective-C, closures were engineered to become degenerate Objective-C objects sufficient to participate in either of the two forms of object memory management: manual reference counting or conservative garbage collection, regardless of their language of origin. The C usage is completely specified by `closure_copy` and `closure_free` as will be discussed and has no reliance on memory management facilities offered by either C++ or Objective-C, or Swift.

In 2014 Apple introduced the Swift programming language which also supports closures. Microsoft supports Objective-C programming for iOS and Windows 10 and is said to be working on Swift support as well.

Closures provide an expressive way to design powerful APIs and have been used extensively to retrofit existing and introduce new APIs. It has been said that one cannot program a Mac, iOS, watchOS, and now tvOS application without them.

An implementation of closures can be found in the open source clang and runtime projects where they are called "Blocks".

Closures were discussed at WG14 in N1370 at the Markham09 meeting and specifically proposed for C11 in N1451 at Florence10. The vote was 6-5-4 in favor, not nearly consensus, and, it was said by abstainers, mostly because it was too late.

Closures have also been presented and discussed in "the early days" of CPLEX, "Closures vs Cilk_spawn" emails of 2013/07/13, and presented as a general alternative to most of what has been worked upon as "task parallelism". A straw poll indicated that closures were indeed very interesting and should be taken up at WG14. The library approach of Apple's was shown to be able to compose arbitrary graphs of synchronization among work units including, and, beyond, fork-join parallelism (as OpenMP can do).

2 Design

The design goals for closures at Apple were expressiveness, completeness, efficiency, and safety.

2.1 Expressiveness

Although closures had been proposed even earlier at Apple, a new line of thinking centered around the ability to quickly capture a function and its arguments for use elsewhere. Whereas Cilk does

```
    Cilk_spawn SomeFunction(expression1, expression2)
```

the idea was that, where ^ is used as a new unary operator,

```
       ^SomeFunction(expression1, expression2)   // never
implemented
```

would direct the compiler to produce a "Cheap Invocation" C data structure of some sort that could be used in many different ways, less expensive and more universal than the existing Objective-C Foundation *invocation* object.

A second idea was that it should be possible to also do partial function evaluation (often confused by me as Currying), where

```
       ^drawline(graphicsContext, _, _, _)      // never
implemented
```

would produce a function that took three arguments and would call `drawline` with a stashed away `graphicsContext`.

The third idea was closures, and since it turned out that it could do the other two ideas as well, it was best, and most expressive, to just have one general concept rather than three.

The syntax evolved to introducing ^ (caret) as a new unary operator and as a closure indicator in a declaration. Caret is used to denote pointer to closure. Much like Church's lambda, the syntax for a function expression is caret, arguments, and body with of course a return type, ,

```
        ^void (void) { SomeFunction(expression1,
expression2); }
```

for the first idea and

```
        ^void (float x, float y, float z)
{ drawline(graphicsContext, x, y, z); }
```

for the partial evaluation example. Like a function object, a closure object does not answer to sizeof, unlike a function, it can not be declared and has no name, it is an anonymous function expression. Instead, a function expression yields a pointer to a closure.

Stating this more directly, function expressions only ever appear after the new unary operator ^ (caret), to wit:

```
         dispatch_async(queue, ^void (void)
{ SomeFunction(expression1, expression2); });
```

Further, if the return type of a function expression can be inferred from the consistent type of return expressions, or lack thereof, the return type specification may be omitted. So the above statement can also be written

```
         dispatch_async(queue, ^(void)
{ SomeFunction(expression1, expression2); });
```

Even better, a closure expression that has an inferred return type that also has no arguments, as is the case here, can be written

```
         dispatch_sync(queue, ^{ SomeFunction(expression1,
expression2); });
```

which is very concise and expressive, especially since this captures the heart of *unit of work*.

Pointers to closures are declared and in a directly equivalent way to pointers to functions by substituting caret ^ for asterisk * in declarations, to wit:

```
        void (^closurePointer)(void) =
^{ SomeFunction(expression1, expression2); };
```

and called, as function pointers can be:

```
        closurePointer();
```

2.2 Completeness

In the previous example, `expression1` and `expression2` were meant to suggest that they could expand to include any variable in scope, and to a certain extent they can and do. However, since the storage duration of an automatic variable in scope when a closure expression is evaluated a `const` qualified version of each variable with automatic storage duration that is named in the expression. Variables of other storage durations are not captured and are referenced directly.

This choice of const capture has two implications. First, it forbids mutation of a captured variable, and second, it disallows communication back to the enclosing scope for those cases where the enclosing scope outlives the closure.

The first restriction is relatively minor. For every const captured variable, a non-const version can be declared in the function expression, initialized with the const captured copy, and mutated at will. This has the advantage of making it very explicit that any mutations in the function expression won't make it back to the enclosing scope. Here's an example:

```
{
     for (int index = 0; index < sizeof(array)/
sizeof(element_t); ++index) {
            iterate_linked_list(list, ^(list_item_t item) {
                   ...
                   int indexCopy = index;
                    while (indexCopy--) {
                         ...
                    }
             });
      }
}
```

The key notion of a true closure is the ability to not only see but share access to automatic variables within its scope. The compromise developed for Apple's closure is to require that the shared variable be declared with a new *shared* storage class. Here's an example that shares between the closure and the enclosing scope:

```
{
     _Shared int maximum = -1;
     for (int index = 0; index < sizeof(array)/
sizeof(element_t); ++index) {
```

```
            iterate_linked_list(list, ^(list_item_ref item)
{
                    ...
                    if (item->member > maximum) maximum =
item->member;
                    ...
            });
      }
      if (maximum > THRESHOLD) { ... }
}
```

Sharing can also occur among many different closures, and that of course you can create closures within closures

```
{
     _Shared int maximum = -1;
     for (int index = 0; index < sizeof(array)/
sizeof(element_t); ++index) {
            iterate_linked_list(list, ^(list_item_ref item)
{
                    ...
                    if (item->member > maximum) {
              maximum = item->member;
              dispatch_async(queue, ^{ ... index; ...;
item; ...; maximum; ... });
                    }

            });
      }
}
```

In this example the call to dispatch_async captures `maximum` and `index` from the outermost scope and `item` from the inner to schedule whatever work on an asynchronous work queue. Each closure in this example differs by the value of the captured `index` and `item` variables.

Quite complicated constructs can be made - one closure can act as a setter for a *shared* variable and another, in the same or related scope, can act as a getter, and yet a third can use the *shared* variable effectively forming a deconstructed object. Closures can reference several *shared* variables.

2.3 Efficiency

Although concurrency was a major goal, a strong minor was usefulness for synchronous uses such as collection iterators where efficiency is paramount. As such, closures and shared variables are allocated in automatic memory - they each live in a compiler written custom structure that has a small header and then the content.

In order to live beyond the lifetime of the enclosing context, it is required that the closure be explicitly copied using a type generic function `_Closure_copy`:

```
    void dispatch_async(queue_ref q, void (^closure)(void))
{
         queue_item_t *item = allocate_queue_item();
         item->closure = _Closure_copy(closure);
          ...
     }
```

and when done, another type generic function must be called to recover the allocated memory resources:

```
         _Closure_free(item->closure);
```

Closures and their closure and shared variable references form a DAG, not a graph, and so a simple reference counting technique allows a closure already in allocated memory to simply (atomically) increment an internal reference counter instead of allocating a redundant further copy for every additional call of `_Closure_copy`. This choice is, of course, up to the implementation. Closures at file scope have no automatic variables to capture and an implementation need not copy them to allocated memory.

A subtle point here is that there is no "this" or "self" name within a closure that references the closure and this ensures that a DAG and not a graph is formed. A graph will have unrecoverable memory resources if copied unless some form of reference cycle garbage collection is employed.

`_Shared` variables that are intended to be accessed from multiple threads shall also be `_Atomic` to avoid data races.

The biggest surprise from the field was the treatment of arrays in automatic storage. They are const copied into the closure. Since stacks other than main are fairly small, this isn't a huge drawback, and for those cases where the array is accessed synchronously (the closure is not copied), a pointer to the array can be formed and the pointer captured. VLAs cannot be captured.

```
{
      point_t array[32];
      ...
```

```
        point_t parray[32][] = array;

        iterate_linked_list(list, ^(list_item_ref item) {
              if (parray[item->index].color == RED) { ... }
        });
}
```

Beyond the concurrency queue library so far discussed, closures are copied and used asynchronously by nearly every Apple library. The Apple GUI libraries use closures to anonymously bind client data to actions instigated by the graphical user interface. The networking libraries use copied closures to asynchronously handle incoming data, termination conditions, and other matters.

This is the general solution that eliminates `void *` parameters that are passed to supplied functions, like in the synchronous `qsort` case, except in asynchronous contexts.

2.4 Safety

An important goal of the work was that any and all closures could be copied and used asynchronously. The programmer can not be prevented, for example, from capturing pointers to local automatic variables and causing undefined behavior, or havoc in other ways. But when the default formulation is safer, efficient, concise, and eloquent, the end result is safer and likely more correct programs, and more quickly as well, and this has to be considered over a swiss-army knife set of options to wound oneself with. This proposal is about making simple things easy and difficult things possible.

3 Implementation

A function expression is straightforward to implement. As with a function, a closure has a return type and an argument list that form its typename, and a list of captured variables with their types, and a list of `_Shared` variables with their types.

Lets discuss the shared variable first. In the example above where there is a `_Shared int maximum`, the compiler rewrites the declaration to be a locally unique structure:

```
struct __shared_1234 {
    struct __shared_variable_header header;
    volatile struct __shared_1234 *stack_or_heap;
    volatile _Atomic int maximum;
} __shared_1234;
```

Once the function expression is parsed, a custom structure is formed for each function expression and, after a common header, the list of captured variables are added to the structure in a const qualified form, and pointers to each `_Shared` variable are declared. The structure is scheduled to be emitted at file scope. Next, the custom closure implementation function is formed using the return type and arguments. A pointer to the custom structure is inserted as a secret first argument with a compiler chosen hidden name `__secret`, and the closure body is rewritten to use `__secret->captured` for every use of a captured variable. `_Shared` variable accesses are rewritten as `__secret->stack_or_heap->shared`, and the custom implementation function is emitted.

Thus, for the `^{ ... index; ...; item; ...; maximum; ... }` expression the compiler would emit at file scope:

```
struct __closure_abcd {
   struct __closure_header header; // common
   const int index;
   const list_item_ref item;
   volatile struct __shared_1234 * __shared_1234;
};

static void __closure_abcd_implementation(struct
__closure_abcd *__secret) {
    ... __secret->index; ...; __secret->item; ... __secret-
>stack_or_heap->maximum; ...
}
```

The compiler does need to do something slightly tricky near the function expression site, and that is to declare the actual closure structure in automatic memory, initialize it, and reference it.

The code becomes

```
     iterate_linked_list(list, ^(list_item_ref item) {
          ...
          if (item->member > maximum) {
               maximum = item->member;
               struct __closure_abcd __closure_abcd =
{ __closure_abcd_implementation, index, item, &maximum };
               dispatch_async(queue, &__closure_abcd);
          }

     });
```

`_Shared` variables are declared in a custom structure with a pointer to itself preceding the variable. When copied, the pointer is reset to the allocated memory copy, and the copy also has its pointer initialized with a pointer to itself, the allocated memory copy. The pointer within the header is declared volatile to ensure that it is reloaded every time it is used, thus guaranteeing that each access sees the latest version and especially that it catches the transition to allocated memory. This is not a data race; it is undefined behavior if a closure is used from a secondary thread without being copied. The copy is done from the original thread synchronously.

4 Proposal

Apple's concurrency library Grand Central Dispatch (GCD) has been open-sourced and is available on Linux and FreeBSD. The Swift programming language is moving to Linux and IBM, and so Apple's closures have, to some degree, become part of the ABI on those platforms.

WG14 is in an excellent position to simply adopt this existing industry practice rather than redesign it, and I strongly recommend this general course of action.

There are, of course, places where WG14 can add value.

4.1 Library

These are possibly new library entry points that could be added. The following three come to mind quickly, particularly since two are already established practice on Apple platforms.

4.1.1 `thrd_create`

In addition to the existing 7.26.5.1 `thrd_create` function

```
typedef int (*thrd_start_t)(void *);
int thrd_create(thrd_t *thr, thrd_start_t_ func, void
*arg);
```

WG14 could add

```
typedef int (^thrd_start_closure_t)(void);
int thrd_create_c(thrd_t *thr, thrd_start_closure_t
closure);
```

which eliminates the need for the programmer to write a function and do casting from `void *` to any parameters of interest.

4.1.2 qsort

In addition to

```
void qsort(void *base, size_t nmemb, size_t size,
     int (*compar)(const void *, const void *));
```

WG14 could, as Apple has done, provide

```
void qsort_b(void *base, size_t nmemb, size_t size,
     int (^compar)(const void *, const void *));
```

so that variations in sorting determined at runtime can be easily encoded in a closure.

4.1.3 bsearch

In addition to

```
void *bsearch(const void *key, const void *base,
     size_t nmemb, size_t size,
     int (*compar)(const void *, const void *));
```

WG14 could, as Apple has done, provide

```
void *bsearch_b(const void *key, const void *base,
     size_t nmemb, size_t size,
     int (^compar)(const void *, const void *));
```

5 Process

C11 is considering creating a TC2 that incorporates the syntax changes and other edits from closed Defect Reports. The syntactic changes, for this proposal, were sketched out in N1451 and need revision. Depending on interest, a further paper can be developed and would provide detailed syntactic changes and constraints. At this point, such detail seems premature.

Blaine Garst