Accessing the Context of Nested Functions

Martin Uecker, Graz University of Technology
2025-07-20

0. Introduction

In this paper, I will summarize how nested functions in the spirit of ALGOL60, Pascal, GNU C, D, and similar languages could be integrated into C. I believe that this is a cleaner approach compared to C++ style lambdas, which are designed for an object-oriented language with templates and may not be an ideal fit for C. Before I begin, I like to stress that I do **not propose a solution that requires any kind of trampolines or executable stack**, but instead a mechanism that makes use of a new wide pointer type as already proposed earlier (N2661 + N2862). For a technical discussion I refer to these documents. Although I do not fully agree with all points and conclusions there, I like to point the reader to the forthcoming document by JeanHeyd Meneide, which includes excellent discussion of important points and alternative solutions that should be considered carefully.

One criticism of nested functions in GCC is that they can not be used after the enclosing functions has ended if the access automatic variables whose lifetime has ended. The main aim of this paper is to show how lifetime extension of such nested functions can be added as a feature. While I am not yet fully convinced that lifetime extension of nested functions is necessarily a good idea in the context of C, I like to make it clear that standardizing a basic version of nested functions will not prevent us from adding a feature for lifetime extension later.

As an example, I will discuss an API with a callback function without and (more importantly) with an additional data argument (e.g. *qsort*, *gtk*, etc.). The examples are simplified and for illustration purposes only and do not reflect the complexity of real-world code.

```
// API with callback using regular function pointer type

typedef int (*cb_t)(int)

void api_old_simple(cb_t);
void api_old(cb_t, void *);
void api_old_copy(cb_t, void *data, size_t data_size);
```

1. No Capture - No Problem

If only automatic variables that are compile-time constants (but not their address), types that are not variably modified, and variables that do not have automatic lifetime are accessed, then conversion to regular function pointer is not a problem and a new type or trampoline is not required. As suggested by Chris Bazley and also used in the D language, we annotate such capture-less nested functions with *static*, but we note that this property can also be inferred by the compiler. For the writing down the examples, we also adopt the syntax proposed for anonymous nested functions (lambdas) from N3645. I general, we omit anonymous nested functions (lambdas) in most of the discussion, but suggest that they should generally have the same semantics as regular nested functions and that it would be mistake to introduce any artificial difference.
Static nested functions can used as shown in the following example.

```
// captures restricted to constants or static variables

int example1()
{
      constexpr int a = 1;
      static int b = 2;

      static int bar(int x) { return x + a + b; }

      api_old_simple(bar); // ok

      // anonymous nested function (lambda)
      api_old_simple((static int(int x)){ return x + a + b; });
}
```

When data needs to be captured, this can be done manually by constructing a structure and passing a *void* pointer to this structure to our API.

```
int example2()
{
      // We have to manually copy the variable and pass a pointer
      // to the data in an argument with type pointer to void.

      int c = 1;
      struct { int c; } data = { c };

      static int bar(int x, void *_d)
      {
            typeof(data) *d = _d;
            return x + d->c;
      }

      api_old_copy(bar, &data, sizeof(data));

      // anonymous nested function (lambda)
      api_old_copy((static int(int x, void *_d)){

            typeof(data) *d = _d;
            return x + d->c;

      }, &data, sizeof(data));
}
```

Static nested functions can already be seen as an improvement, but **their usage is not always type safe**. In addition, they require writing boilerplate code to initialize a structure and to access the variables with the help of the *void* pointer.

2. Type Safe Callbacks

Already in ALGOL 60, a solution was proposed: Nested functions that can access variables in their non-local environment directly, i.e. by name. A common implementation technique for this is to pass a static chain in a hidden argument, for which there usually exists a register in the platform ABI which is used for this purpose by many other languages. Because on most platforms the native function pointer type can not store the static chain and this would require run-time creation of a trampoline that loads the static chain register before calling the real function, we recommend to instead introduce a new wider pointer type as proposed in N2862. With such nested functions, the code can then be written in a **type safe and convenient** way as shown in the following example.

```
// API with new wide function pointer

typedef int (*cb_wide_t)(int) _Wide; // N2862

void api_new(cb_wide_t);

void example3()
{
      int d = 4;

      int bar(int x) { return x + d; }

      api_new(bar);

      // anonymous nested function
      api_new((int(int x)){ return x + d; });
}
```

Please note that this code is **compatible with nested functions as implemented in GCC.** GCC could start to support conversion of nested functions to the wide function type without creation of a trampoline, while continuing to support conversion of the pointer to a regular pointer as an extension with the use of a trampoline. It could be expected that this is done for a transitional period only and at some point in the future deactivated as default in newer language modes - in this way also eliminating the use of trampolines in GCC. In fact, conversion to regular functions pointer types is an orthogonal design aspect. **Consequently, the existing GCC feature does not prevent us from introducing nested functions with the same syntax.**

```
void example4()
{
      int d = 4;

      int bar(int x) { return x + d; }

      api_old(bar); // GNU extension, implementation-defined
      api_new(bar); // ok
}
```

3. Compatibility with Existing APIs

When introducing a new type, there is a need to pass nested functions to legacy APIs that do not yet make use of it. N2862 proposes a basic solution for this using the wide pointer library API. Another possibility is to combine the use of a capturing and static (non-capturing) nested function.[1] This can be achieved using an explicit static (!) trampoline as in the following example.

```
void example5()
{
      int d = 4;

      int bar(int x) { return x + d; }

      // static (capture-less) nested function

      static int trampoline(int x, void *ptr)
      {
            return (*(cb_wide_t*)ptr)(x);
      }

      api_old(trampoline, &(cb_wide_t){ bar });
}
```

This workaround then allows the use of nested functions with an legacy API that requires regular function pointers. As a downside, this still requires writing boilerplate code and involves an indirect call. A better solution might be to add explicit support for passing the static chain via a traditional *void* pointer argument. Here, I propose the *_Closure(data)* syntax to indicate that the static chain should be loaded from the specified argument (at most a single *mov* instruction)[2], and then propose to reuse the same keyword to give the user direct explicit access to an object that represents the environment.

```
void example6()
{
      const int d = 4;

      // static chain passed via specified argument

      int bar(int x, void *data) _Closure(data)
      {
            return x + d;
      }

      api_old(bar, &_Closure(bar));
}
```

1 I saw this idea first in an example by JeanHeyd Meneide where the non-capturing lambda was used to call a C++-style lambda with anonymous type.

2 This was suggested a while ago, but I forgot by whom. My apologies for not being able to give credit.

4. Lifetime Extension: Copying Values

We have now all the ingredients to discuss the main part of this paper. Here, I will show how we can extend this approach to enable lifetime extension by copying the environment similar to how it is done in C++. First, to be able to copy the environment we need to capture the variables by value. We recommend to allow this only for variables that are *const*-qualified. For *const*-qualified variables there is no semantic difference to variables that are accessed by name. Hence, **a capture annotation to distinguish capture by value and capture by name is not needed.** It is also seems safer to require *const* qualification as it avoids any doubt which value is used when the original and the copy that shadows it get out of sync (cf. the quiz in the appendix). To this end, the explicit use of *_Closure* is restricted to nested functions that capture only *const*-qualified variables.

```
void example7()
{
      // const-qualified variables can be copied

      int (*const p)[10] = malloc(sizeof *p);
      if (!p) return;


      int bar(int x, void *data) _Closure(data)
      {
            return (*p)[x];
      }

      // sizeof can be used to obtain the required size

      api_old_copy(bar, &_Closure(bar), sizeof(_Closure(bar)));
}
```

This solution can now be used in *api_data_copy* to copy the captured environment around. Unfortunately - and similar to the proposals that adopt C++-style lambdas in C - this is still severely limited! The captured pointer is now hidden inside the nested function with no way to access it from the outside. To be able to free the memory, one would need to maintain a separate copy of the pointer in an additional data structure[3], and similar considerations apply when one wants to do a deep copy of more complicated data structures such as a tree. While this might be good enough for some applications, I fear that the need to maintain a separate copy of the captured pointers would defeat the point of removing pointless boilerplate code in many others. Remember, the only reason why one might want to introduce nested functions with capturing and possibly lifetime extension in the first place is to avoid constructing and copying data structures manually. If this falls apart outside of toy examples, this whole enterprise may not actually be worth it.

3 Or use other workarounds such as allowing only one invocation that frees the memory at the end or requiring an additional cleanup mode activated with an additional parameter.

5. Lifetime Extension: Exposed Captures[4]

In the following I will sketch a way out that is still relatively simple. Internally, *_Closure(bar)* is a structure that holds the copied values. Instead of just exposing its size, we could expose this a structure type to the user, also exposing its members. One can then pass it to a static (capture-less) nested function that can inspect the type and free the pointer as shown in the following example.

```
void api_old_copy_del(cb_t cb, void *data, size_t size,
                      void (*del)(void *));

void example8()
{
      int (*const p)[10] = malloc(sizeof *p);
      if (!p) return;

      int bar(int x, void *data) _Closure(data)
      {
            return (*p)[x];
      }

      // static nested functions acting as destructor

      static void del(void *_data)
      {
            // the structure type is visible at this point

            typeof(_Closure(bar)) *data = _data;
            free(data->p);
      }

      api_old_copy_del(bar, &_Closure(bar), sizeof(_Closure(bar)), del);
}
```

I want to discuss some limitations of this approach. First, in some cases one might want to have mutable captures, *e.g.* to be able to track whether a pointer was freed or not. Another is issue is that we now have two nested functions, where the second one still has to jump through hoops to access the pointer. Why can't they simple access the same pointer – sharing the environment? We could simply capture the pointer in the destructor, which might be acceptable in simple examples. In general, we would have two copies of the environment which is wasteful for a larger number of captures or larger objects and could easily cause confusion when one pointer is modified or freed but the other copy is not. Note that this sharing works in Apple's Blocks extension (N2030), but at the cost of having a run-time infrastructure that maintains lifetimes using reference counting and that copies variables automatically to the heap. Finally, I want to point out that alignment requirements of captured variables can also cause complications when the environment is copied to memory that has insufficient alignment.

4 As far as I know, this idea was mentioned on the WG14 reflector in a discussion with Alex Celeste. It is now also used in JeanHeyd's forthcoming proposal to address these concerns.

6. Another Idea: Shared Environments

Apple's Blocks extension lets us work with captures that are shared between different nested functions providing a much nicer solution. In general, the possibility to have shared captures is very useful. One possibility would be to capture the shared environment of all nested functions in the same enclosing function simultaneously into a single capture object. In this case, we could drop the requirement that the captured values are *const*-qualified, because the environment is shared between all functions and they then all operate on the same instance of each variable - until a complete copy of the environment is made for all related functions together. As shown in the following example, this would make writing required helper functions such as destructors much easier.

```
void api_old_copy_del(cb_t, void *data, size_t size, void (*del)());

void example9()
{
      int (*p)[10] = malloc(sizeof *p);
      if (!p) return;

      int bar(int x, void *data) _Closure(data)
      {
            return (*p)[x];
      }

      // nested function acting as destructor

      void del() { free(p); p = NULL; }

      api_data_copy_del(bar, &_Closure(bar), sizeof(_Closure(bar)), del);
}
```

While this is just an initial sketch which leaves many questions open, it is a promising way forward that should be explored further. The main open question is what happens in case of multiple nested scopes (or functions). In principle, it would be nice to also make the tree structure of the environments directly accessible to the user, which would enable implementation of Blocks-like functionality on top of it. This motivates a final feature that allows reading and updating an implicitly captured static chain, *e.g. _Closure(bar).chain* = ..., but this is left for future work.

7. Polymorphic Types for Safety

A remaining issue with the previous approaches is that we the need to pass the environment as a *void* pointer, which is not type safe. Here, the key insight is that the reason for this is that we need to erase a type to be able to use a generic API. In general, this use case can be addressed with polymorphic types (N3212). Consequently, we can enhance our API using polymorphism where *_Closure(T)* now implies that the captured environment has a type *T*. This notation could replace the *_Wide* pointer syntax we had previously.

As the following example shows, this already looks quite nice and is type safe, as it the type information could make sure that the right destructor is used for each nested function.

```
void api_poly_copy_del(_Type T,
     int (*task)(int) _Closure(T),
     void (*del)() _Closure(T));

void example10()
{
     int (*p)[10] = malloc(sizeof *p);

     int bar(int x) { return (*p)[x]; };
     int del() { free(p); p = NULL; };

     api_poly_copy_del(_Typeof(_Closure(bar)), bar, del);
}
```

With some suitable mechanism for inference of type arguments this could become even nicer.

```
void example10()
{
     int (*p)[10] = malloc(sizeof *p);

     int bar(int x) { return (*p)[x]; }
     int del() { free(p); p = NULL; }

     api_poly_copy_del(*, bar, del);
}
```

The *api_poly_copy_del* function could then copy the environment by extracting the closure type from the pointer, which it can do because it is parametrized by the type T[5], and could then pass it to another thread for asynchronous processing, for example. Passing a type also resolves the alignment issue.

```
void api_copy_poly(_Type T,
     int (*task)(int) _Closure(T), void (*del)() _Closure(T))
{
     _Var(T) *p = aligned_alloc(alignof *p, sizeof *p);
     if (!p) return;
     *p = _Closure(*p);

     // start other thread and pass copy etc.
     ...
```

5 A possible improvement suggested by Thiago Adams for lambdas would allow recovering the size from the closure itself – removing the need to pass size or type explicitly, but this would imply an additional requirement imposed onto the ABI of wide pointers. I like to avoid such a requirement to remain compatible with any other language and implementation that have closures or callable objects.

8. Conclusion

The main conclusion of this work is that we can build the required features incrementally and systematically. The steps described in Sections 2 – 3 are relatively straightforward and are based on existing practice. Syntax to set the static chain from an argument (Section 4) seems to be a very simple and logical extension. From the discussion of lifetimes in Section 5 and 6 it should become clear that there are many possible design choices and also still many open questions. Just like the alternative proposals, this part is not rooted in existing practice. Nevertheless, the important message is that adding nested functions as used by many other languages would not prevent us from adding a feature for lifetime extension later. Section 5 shows explicitly how this could be done. Section 6 then shows that there may also be promising alternative approaches that haven't been sufficiently explored at this point. Finally, Section 7 shows that integration of other language features such as polymorphic types could open new possibilities that should be considered.

In summary, it is a strategically safe choice to proceed with the steps outlined in Sections 2 - 4 while we should wait with any feature for lifetime extension until a fully convincing design emerges and until there is practical experience with a prototype.[6]

9 References

- N2030 Garst, A Closure for C, 2016-03-21
- N2661 Uecker, Nested Functions, 2021-02-13
- N2862 Uecker, Function Pointer Types for Pairing Code and Data, 2021/11/30
- N2924 Gustedt, Type-generic lambdas v5, 2022-01-31
- N3212 Uecker, Polymorphic Types, 2024-01-14
- N3545 Adams, Literal functions, 2025-07-13

- Backus et al. (ed: Naur), Revised Report on the Algorithmic Language ALGOL60, 1962
- GCC Project, Manuel of the GNU Compilers: Nested Functions, accessed 2025-07-19
- D Language Foundation, Language Reference: Functions, accessed 2025-07-19
- Meneide, Functional Functions - A Comprehensive Proposal Overviewing Blocks, Nested Functions, and Lambdas for C, 2025/07/11 (initial version)

6 I believe that for any language feature adopted in ISO C there should be real usage experience. It is still noteworthy though that the ALGOL60 report already includes a convincing (but untested) use case in the form of an implementation of a Runge-Kutta method that makes use of nested functions. For a contemporary and actively maintained project using nested functions in C, see BART: https://github.com/mrirecon/bart For BART, no concerns related to lifetime or capture by name can be reported based on almost a decade of experience with this feature.

Appendix A: List of Concerns with C++-style Lambdas in C

General issues shared with approaches for nested functions in C:

- Passing of a lambda as a regular parameter needs either trampolines or a new type.
- Lambdas with lvalue capture suffer from the same lifetime limitations as nested functions.

```
auto foo(int i) { return [&](){ printf("%d\n", i); }; }
```

- Not having destructors and smart pointers in C requires workarounds not needed in C++.
- Not having explicit access to the structure holding the captured values requires unsafe byte copies, causes issues with alignment and makes deep copying impossible.[7]

Specific problems of C++-style lambdas or similar designs:

- Making the lambda itself have a unique anonymous object type in C means it can only be invoked immediately which seems useful only in macros. In C++ it can be returned from and passed to template functions.
- To address the various use cases, there are many different ways to capture variables [&], [=], [], [a], [&b], [a = b], mutable, etc. adding a lot of complexity that does not seem necessary.
- Value captures can be confusing as they shadow the original variable under same name.
- Copying of the environment makes it more difficult to share state between different nested functions, but this seems desirable in some applications.
- The syntax for capture is much more complex than needed.
- Other features from C++ may need to be pulled in from C++ to make them fully useful, such as trailing return types, return type deduction, and generic arguments.
- Adopting lambdas from C++ would limit out design freedom relative to C++. We can not easily change specific aspects when it might be better for C, because it would then be a divergence from C++ that should be avoided and will be opposed by implementers.

Appendix B: Quiz: What does the following C++ program print?

```
#include <stdio.h>

int j = 3;

int main()
{
      int i = 3;

      auto foo = [=](){ printf("%d\n", i); };
      auto bar = [=](){ printf("%d\n", j); };

      i = j = 4;

      foo();
      bar();
}
```

7 That these issues were missed in initial proposals for lambdas in C shows that experience in C++ is not sufficient to identify problems relevant to C.