WG14 N2661

Title: **Nested Functions**
Author: **Martin Uecker, University Medical Center Göttingen**
Date: **2021-02-13**

1. Introduction

Many programming languages support nested functions including ALGOL, Pascal, Modula 2, Oberon, Ada, Python, Swift, D, Javascript. Nested functions are useful as helper function, for callbacks, as generators, for higher-order programming, and much more. Implementation techniques for them are well known [1]. Some C compilers support nested functions as an extension [2,3] and these implementations are similar enough to successfully use a useful common subset by using macros to hide syntactic differences (although with some limitations and incompatibilities). This paper proposes to add simple nested functions to C. Full first-class support for nested functions requires function pointers that pair code and data. This requires special consideration because of existing ABI constraints. We will discuss basic concepts, design questions, and implementation techniques.

Example 1a (Pascal):

```
function foo(x: real): real;
    function bar(y: real): real;
    begin
        bar := x + y
    end;
begin
    foo := bar(3)
end;
```

Example 1b (Python):

```
def foo(x):
     def bar(y):
          return x + y
     return bar(3)
```

Example 1c (GCC and Clang with macro):

```
double foo(double x)
{
     NESTED(double, bar, (double y)) { return x + y; };

     return bar(5);
}
```

Example 1d (GNU C):

```
double foo(double x)
{
      double bar(double y) { return x + y; }

      return bar(5);
}
```

Nested functions are a well understood concept with clearly defined semantics. Basic syntax and semantics follow directly from existing language rules. Simply by allowing nested function definitions to appear inside other functions Example 1d is already fully defined **without any other change to the language.** In particular: 1. No new syntax is required for nested functions. 2. Existing scoping rules hold for all identifiers. 3. Existing lifetime rules hold for all objects. These properties make nested function easy to learn, teach, and analyze. A couple of additional constraints are needed to define the interaction with other parts of the language, i.e. storage classes, linkage, and control flow.

2. Special Semantic Aspects of Nested Functions

Nested functions are in many ways just like regular functions and most language rules work without any change. Nevertheless, allowing functions to be defined in nested scopes also introduces some fundamentally new aspects which require careful consideration.

2.1 Accessing Variables of Enclosing Functions

The usual scoping rules imply that nested functions can access the variables for their lexically enclosing functions. By allowing this, nested functions become more powerful, because they can abstract away context and introduce helper functions that specialize to a local use case.

Example 2:

```
struct bar { double data[4]; };

double foo(struct bar x[10])
{
      double access(int y) { return x[y].data[3]; }

      return access(3) + access(5) – 2. * access(4);
}
```

If the nested function is not inlined and compiled into a separate function, some mechanism is required to be able to access the variables of the enclosing function. We will discuss the traditional implementation techniques in Section 3.1.

2.2 Taking the Address of a Nested Functions

In C it is possible to take the address of a function and pass it on to other functions. Functions therefor are first-class citizens of the language which enables higher-order programming. If this is possible also for nested functions, then this further enables improved higher-order programming. Consider the following example where a comparison function for a sort function (similar to `qsort`) depends on a parameter:

Example 3a (sort with void pointer)

```
struct my_element { int val; };
struct my_data { int mod; };

static int cmp_modulo(const void* va, const void* vb, void* vdata)
{
      struct my_data* data = vdata;
      const struct my_element *a = va, *b = vb;
      return ((a->val % data->mod) - (b->val % data->mod));
}

void sort_modulo(int mod, int N, struct my_element array[N])
{
      struct my_data data = { mod };
      sort(array, N, sizeof(array[0]), cmp_modulo, &data);
}
```

Here, the dependency of the comparison on the variable `mod` requires passing around a void pointer through `sort` to the comparison function. Similar code is often seen in C programs in various similar scenarios (e.g. callbacks), and the corresponding loss of type safety if often considered a major disadvantage of C. As the following example demonstrates, nested functions enable an improved type-safe design and usage of higher-order interfaces.

Example 3b (sort with nested function)

```
struct my_element { int val; };

void sort_modulo(int mod, int N, struct my_element array[N])
{
      int cmp_modulo(int a, int b)
      {
            return ((array[a].val % mod) - (array[b].val % mod));
      }

      sort(array, N, sizeof(array[0]), cmp_modulo);
}
```

When nested functions access the variables of their enclosing functions, taking the address of the nested function then requires the creation of a function pointer that is a pair of a code and a data (environment) pointer. On some platforms this is difficult to implement without breaking their existing ABI. We will discuss different implementation options in Section 3.2.

2.3 Returning the Address of a Nested Functions

In certain cases it could also be useful to return the address of a nested function.

Example 5:

```
typedef int (*adder_f)(int x);

adder_f make_adder(int x)
{
      int addx(int y) { return x + y; }

      return addx;
}
```

The code in this example is well-defined when applying existing rules, but the nested function - when called later by some other function - would access a variable that went out of scope. When applying the lifetime rules of C any access to variables whose lifetime has ended invokes undefined behavior. If a nested function does not reference a variable of an enclosing function there seems to be no real reason not to treat identical to a regular functions defined at file scope. Thus, applying existing language rules does not allow Example 5 to work as intended, but yields reasonable semantics for many simpler cases. To be compatible with established implementation techniques (see Section 3) it is necessary to directly restrict the lifetime of nested functions: The lifetime could be bounded by the lifetime of any parent function that contains variables the nested function refers to by name.

A future extensions could then also allow the creation of closures that capture variables to let them escape the local context as required in Example 5 (cf. the Blocks extension n[3,4]). As this goes beyond what can be implemented using stack-based automatic, this is not discussed further in this document which aims to describe only a basic framework for nested functions.

2.4 Linking

Linking the identifier of a nested function to an identifier at file scope would make it possible to call a nested function even when its enclosing parent is not active. This seems undesirable and can be prevented by prohibiting storage class specifier and specifying that nested functions have no linkage. (Note hat GNU C allows forward declaration of a nested function using the storage class specifier `auto`, but we did not include this here.)

2.5 Non-local Jumps

While jumps into a nested function do not appear to make sense and should be forbidden, non-local jumps out of a nested function into enclosing function have well-defined semantics and should be allowed. In fact, this functionality this is very similar to `setjmp` and `longjump`, but without many of its disadvantages. Non-local jumps are supported in GCC (and also in a simple proof-of-concept compiler developed by the author). We will touch upon implementation questions in Section 3.3.

Example 6a (longjmp)

```
void bar(int a, jmp_buf err_env)
{
      int err = 0;

      // do something, maybe set err

      if (err)
            longjmp(err_env, err);

      // do something
}

int foo(void)
{
      jmp_buf err_env;
      int err = 0;

      if (err = setjmp(err_env))
            goto error_out;

      bar(2, err_env);

      // do something

error_out:
      return err;
}
```

Example 6b (nested function)

```
typedef void (*error_cb_t)(int);

void bar(int a, error_cb_t err_cb)
{
      int err = 0;

      // do something, maybe set err

      if (err)
            err_cb(err);

      // do something
}

int foo(void)
{
      int err = 0;
      void error_cb(int x) { err = x; goto out;  }

      bar(2, error_cb);
out:
      return err;
}
```

Non-local jumps out of nested functions have several advantages over `longjmp`:

1. Control flow is easier to analyze because it is not mediated by the jump buffer `env` whose state would need to be tracked by a reader (or compiler). Instead, source and target of the non-local jump can be directly read off the code using the same syntax also used for other jumps.

2. The interface is not limited to passing a single integer back, but can use local variables of the parent function for communication.

3. Nested functions can be used by compilers targeting C to implement features that require run-time stack access [6]. This would further strengthen C's important role as an intermediate language by addressing a current limitation.

4. The state of all variables is always well-defined without the requirement to use volatile.

2.6 Lambda Expressions

Lambda expressions are anonymous nested functions defined inside other expressions. They require additional new syntax and semantics beyond what is discussed in this paper. Using the syntax C++'s lambda expressions, Example 3a could be written as follows.

Example 7a (C++ lambda expressions)

```
struct my_element { int val; };

void sort_modulo(int mod, int N, struct my_element array[N])
{
      sort(array, N, sizeof(array[0]), [&](int a, int b) -> int
      {
            return ((array[a].val % mod) - (array[b].val % mod));
      });
}
```

In contrast to conventional named nested functions, lambda expressions require additional new syntax, the new ability to parse compound statements inside other expressions, and generally require more effort in specification and implementation. Lambda expressions can be very useful for type-generic macro programming, where regular nested functions are less useful because they can not be used inside an expression (or only with further language extensions such as compound expressions). An advantage of lambda expressions is that they are already supported by C++ and a specific proposal to add some limited form of lambda expressions to C was recently put forward in a series of papers [7-10]. Here, we do not further discuss lambda expressions, put point out that both features are complementary and should not be seen as mutually exclusive. It is suggested that it may be prudent to first introduce named nested functions as the simpler and better understood version of nested functions before extending it to lambda expressions.

3. Implementation of Nested Functions

Nested functions are a very well-known concept and implementation strategies are described in text books [1]. We will briefly discuss some relevant points.

3.1 Accessing Variables of Enclosing Functions

The traditional way to implement access to variables in an enclosing function is briefly described in the following. The nested function receives as a hidden argument an environment (static chain) pointer that points to the stack frame of the lexically enclosing function [1]. Using this environment pointer it is then possible to access the variables stored in the stack frame of the parent function (and by following the pointers also from all ancestors in case of deeper nesting). As many programming languages use nested functions, the ABI for passing an environment pointers already exists for most (if not all) platforms. An alternative implementation strategy is to copy pointers to all referenced variables into a new structure and pass a pointer to this structure to the nested function (this is similar to what C++ does). Note that because nested function are always private to the enclosing function(s), there is no ABI that needs to be defined for accessing variables of parent functions. Each compiler can use different techniques or even mix different techniques inside the same program.

3.2 Taking the Address of a Nested Function

Being able to take the address of a nested functions is more difficult, because calling the function later needs knowledge of the environment pointer. Thus, simply taking the address of the code is not enough, ABI support is required for pointer types that include both code and the environment pointer. On some platforms this is not a problem, because function descriptors are used that already include an environment pointer. In general, the C standard already allows using a wider function pointer type that includes an environment pointer. Unfortunately, changing the pointer type would be an ABI breaking change on some platforms were function pointers now have the same size of void pointer (which is also a POSIX requirement). Various implementation techniques exist to support pointers to nested functions in a backwards compatible way on these platforms (e.g. trampolines [5]), but they all seem to have different drawbacks which make them appear unsuitable as a generic solution (executable stack, backwards compatibility issues, run-time overhead, etc.). Hence, the simplest and most promising alternative appears to be the introduction of a new extended function type. Such a type has a variety of other use cases and would bridge a glaring gap in interoperability with and among many other languages that have nested functions or callable objects (e.g. C++). In its simplest form, a pointer to an extended function could simple be a pair of a code and a data pointer where calls using such pointers would then load the data pointer into the static chain register specified by the ABI before calling the code.

3.3 Non-local Jumps

Non-local jumps require resetting the stack pointer to the right value. As this is the same problem as finding the variables of an enclosing function it can be solved in the same way.

4. Required Wording Changes

The required changes for the text of the C standard are very small. In addition to the (trivial) syntax change and renaming of Section 6.9, the following preliminary wording changes should be sufficient to introduce nested functions. To avoid ABI issues, an extended function type should also be introduced at the same time. As this is useful independently for language interoperability this is not included here but will be proposed in a future paper. As C does not impose many requirements on function pointers, adding such a type is relatively simple and mainly requires amending the rules for compatibility, conversion, and comparison of pointers of that type.

6.2.2 Linkages of identifiers

6 The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier extern, **a block scope identifier for a function definition.**

6.8.6.1 The goto statement

Constraints

1 The identifier in a goto statement shall name a label located somewhere in the enclosing function. A goto statement shall not jump from outside the scope of an identifier having a variably modified type to inside the scope of that identifier**, and shall not jump from the outside into the body of a function.**

6.8.4.2 The switch statement

Constraints

2 If a switch statement has an associated case or default label **within a function or** within the scope of an identifier with a variably modified type, the entire switch statement shall be within **that function or** scope of that identifier**, respectively.**

6.9.1 Function definitions

Constraints

4 The storage-class specifier, if any, in the declaration specifiers shall be either extern or static. **A function definition at block scope shall not include a storage-class specifier.**

13 The lifetime of a function that is defined at block scope and that references by name an object of automatic storage duration defined in an enclosing function ends when execution of any such enclosing function ends. Lifetime of all other functions is the entire execution of the program. A pointer to a function becomes indeterminate when the lifetime of that function ends.

5. Conclusion

Nested functions fit well into the C languages. They are simply to specify and easy to implement while substantially improving the language by enabling type safe higher-order interfaces without void pointers, providing a better alternative to `longjmp`, and strengthening C's role as an intermediate language targeted by compilers of other languages. The introduction of a new extended function type is suggested to allow existing platforms to implement nested functions in a backwards compatible way.

6. References

1. Aho AV, Lam, Monica S-S; Sethi R, Ullman JD (2006). Compilers: Principles, Techniques, and Tools (2 ed.). Boston, Massachusetts, USA: Addison-Wesley. ISBN 0-321-48681-1.
2. https://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html
3. https://clang.llvm.org/docs/BlockLanguageSpec.html
4. Garst B (2016). *A Closure for C*,. WG14 N2030.
5. Breuel TM (1988). *Lexical Closures for C++*.. USENIX C++ Conference Proceedings.
6. Yasugi M, Hiraishi T, Yuasa . (2006) Lightweight Lexical Closures for Legitimate Execution Stack Access. In: Mycroft A., Zeller A. (eds) Compiler Construction. CC 2006. Lecture Notes in Computer Science, vol 3923. Springer, Berlin, Heidelberg.
7. Gustedt J (2021).WG14 N2632
8. Gustedt J (2021) WG14 N2633.
9. Gustedt J (2021) WG14 N2634.
10. Gustedt J (2021) WG14 N2634.