--------------------- N3678 Local functions ---------------------------

Author : Thiago R Adams
Date   : 2025-09-30
Project: ISO/IEC JTC 1/SC 22/WG 14
Title : Local functions
Target audience: Implementers, users
Prior art: GCC Nested functions without capture

ABSTRACT

This proposal introduces support for defining functions inside another function, similar to the GCC extension nested functions. Unlike GCC nested functions, local functions do not have full access to the variables of the containing function.

1. INTRODUCTION

C currently allows function declarations inside block scope, but not function definitions.

Allowing inner function definitions would enable:

- Better organization of code.

- Encapsulation of helper logic that is not meaningful outside the enclosing function.

- Complements function literals N3679

Following the same principles as N3679, this proposal avoids introducing closures (captures), keeping the semantics simple and compatible with C's execution and memory model.

2. SYNTAX AND SEMANTICS

2.1 Syntax

```
block-item:
  ...
  function-definition
```

Local function definitions are permitted within functions in the same places where variable definitions are allowed; that is, within any block, mixed with other declarations and statements.

The syntax we are proposing is the same as GCC nested functions; however, we have some design questions.

Consider this sample:

```
   void f() { /* external */ }

   int main() {
       void f();   /* local declaration */
       void f() {  /* local definition */
           /* ... */
       }
       return 0;
   }
```

We need a way to disambiguate the declaration of the local function 'f' from its external declaration. GCC uses auto for this purpose.

```
   /* GCC nested function sample */

   void f() { /*extern*/ }

   int main() {
         auto void f(); /*local*/
         void f() { }   /*auto is optional here*/
         return 0;
   }
```

An option is to adopt the same syntax as GCC nested functions and rely on auto for disambiguation. In this case, when GCC nested functions do not capture variables, they already follow the same semantics and syntax as local functions.

The reason we are not introducing captures is the complexity it brings with lifetimes and compatibility with normal function pointers, as well as concerns about the way GCC nested functions work. GCC implements taking the address of a nested function using a technique called trampolines. This technique was described in Lexical Closures for C++ (Thomas M. Breuel, USENIX C++ Conference Proceedings, October 17-21, 1988). [6]

There are security risks associated with trampolines because they involve executable code on the stack. Many modern systems mark the stack as non-executable to prevent exploits such as buffer overflows. [7]

Trampolines are only required if the nested function captures variables. However, internally, GCC may still use trampolines even if variables are not captured, which remains a source of security concerns.

For instance, the code below does not have captures. Compiling this code with -Wtrampolines will confirm the presence of trampolines.

```
/* Use -Wtrampolines in GCC */
#include <stdio.h>

int main() {
    void local() { printf("hello"); }
    void (*f)() = local;
    f();
    return 0;
}
```

https://godbolt.org/z/dT87xEsq7

Adding the -O1 optimization option removes both the trampoline and the warning.

One alternative design for local functions would be to require the storage qualifier static explicitly, or to make its usage optional if the programmer wishes to mark the function as non-capturing. This would, for instance, prevent accidental captures in GCC. The same qualifier could then be used in the corresponding declarations.

```
   void f() { /*extern*/ }

   int main() {
       static void f();    /* local */
       static void f() { } /* static optional or mandatory? */
       return 0;
   }
```

In practice, 'auto' would not be allowed in local function declarations and definitions. If 'static' is optional in function definitions, then function definitions would have the same syntax as GCC when 'auto' is not used. Function declarations would always differ.

Alternatively, we can continue using 'auto', and local functions would not differ in syntax from GCC nested functions. The only difference is that captures would not be allowed for standard local functions, and GCC nested functions would become an extension concerning only captures. This option is also related to N3579, where 'auto' would no longer serve as a storage qualifier. [5]

Captures for local functions and function literals are also not a closed subject, and this design is still open in many ways. Using 'static' would restrict the design of local functions in its current form, and new options could introduce a new qualifier or reuse the old 'auto'.

The static storage qualifier was also proposed in N3654 - Accessing the Context of Nested Functions [3]- which presents alternatives for nested functions without trampolines.

2.2 Semantics

Local functions have semantics similar to those of function literals [1], as described in N3679.

A local function can access all variables of the containing function that are visible at the point of its definition. However, the use of these variables is restricted so as not to depend on their lifetimes.

Tags, enumerators, and functions declared in the enclosing scope are visible and can be used in the return type, parameters, and body of the local function.

```
int main() {

   void f();
   enum E {A};

   enum E local(enum E arg)
   {
      enum E e = A;
      f();
      return e;
   }
}
```

Labels from the enclosing scope are NOT visible inside the local function body.

```
   int main() {
      L1:;
      void local(void)
      {
         /* error: label 'L1' used but not defined */
         goto L1;
      }
   }
```

(A GCC nested function can jump to a label inherited from a containing function, provided the label is explicitly declared in the containing function using __label__)

VM types from the enclosing scope can be used only in the return type and parameters of the local function and are not allowed inside the local function body.

```
   int f(int n) {
      int ar[n];
      void local()
      {
        typeof(ar) b; /* error */
      }
   }
```

Objects with automatic storage declared in the enclosing scope, and which are not VLA types, can be used within the return type, arguments, and inside the function body, provided they appear only in discarded expressions (expressions whose result is ignored).

Samples:

```
      int main() {
         int i = 0;
         void local()
         {
            int j = sizeof(i); // ok
         }
      }

      int main() {
         int i = 0;
         void local() { i = 1; /* error */ }
      }
```

Objects with static storage duration declared at file or enclosing
scope are visible and can be used in the return type, arguments,
and body of a local function.

```
int g;
int main() {
   void local(void) { g = 1; /* ok */ };
}

int main() {
   static int i = 0;
   void local() {
      i = 1; /* ok */
   }
}
```

The value of __func__ is an implementation-defined null-terminated
string when used inside local functions. For comparison, GCC
returns the name of the function.

A type declared in the result of a local function has the enclosing
scope, either block or file scope.

A type declared within the parameter list of a local function has
block scope, which is the local function body itself.

Sample:

```
int main() {

  struct X { int i; } local(struct Y *y)
  {
     struct X x = {};
     return x;
  }

  struct X x; /* OK */
  struct Y y; /* error */
}
```

3. COMPATIBILITY AND IMPACT

Compatibility and impact were already discussed in the syntax section, where the main question is the interaction with existing GCC nested functions.

4. EXISTING IMPLEMENTATIONS

- GCC nested functions without capturing variables.

- Cake http://thradams.com/cake/playground.html
  (Missing some details like VM types)

5. REFERENCES

1 N3645 N3679 - Function Literals
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3645.pdf
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3679.pdf

2 N3657 - Functions with Data - Closures in C
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3657.htm

3 N3654 - Accessing the Context of Nested Functions
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3654.pdf

4 GCC - Nested functions
https://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html

5 3579 - auto as a placeholder type specifier, v2
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3579.htm

6 Lexical Closures for C++
http://www-cs-students.stanford.edu/~blynn/files/lexic.pdf

7 Getting around non-executable stack (and fix)
https://seclists.org/bugtraq/1997/Aug/63

8 N2661 Nested Functions
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2661.pdf

6. WORDING

The wording for this proposal has not yet been provided, as it has not yet been voted on for direction.

7. ACKNOWLEDGEMENTS

I would like to recognize the following people for their help in this work: Martin Uecker, Alejandro Colomar, Jens Gustedt.