N3645 - Literal functions

Author : Thiago R Adams
Date : 2025-07-11
Project: ISO/IEC JTC 1/SC 22/WG 14
Title : Literal functions
Target audience: Implementers, users
Prior art: C++ lambdas without capture
Revision: r0

ABSTRACT

This proposal introduces literal functions into the C language, offering a syntax for defining functions at the point of use. Literal functions allow programmers to define small, local functions without requiring separate file-scope declarations.

1. INTRODUCTION

C currently lacks a mechanism to define functions at the point of usage which leads to:

- Reduced code locality, forcing readers to jump between unrelated parts of a source file.

- The file scope may become cluttered with functions and associated struct declarations used to support captures.

2. MOTIVATION

Many standard C library functions (e.g., 'qsort', 'thrd_create') and common APIs rely on callbacks. Today, using them requires extra boilerplate, especially for asynchronous callbacks.

```
static int cmp(const void *a, const void *b)
{
       return (*(const int*)a - *(const int*)b);
}

qsort(arr, n, sizeof(int), cmp);
```

With literal functions, the programmer could write:

```
qsort(arr, 20, sizeof(int), (int (const void *a, const void *b)
{
      return (*(const int*)a - *(const int*)b);
});
```

Asynchronous callbacks would have a huge improvement in clarity.

Consider this sample:

```
   void async(void (*callback)(int result, void * data), void * data);

   struct capture { int value; };

   static void async_complete(int result, void * data) {
      struct capture * capture = data;
      free(capture);
   }

   int main() {
      struct capture * capture = calloc(1, sizeof * capture);
      async(async_complete, capture);
   }
```

Given the current state of the language, the function async_complete, which is used only once, must be declared at file scope. Since it uses the struct capture, that struct also needs to be declared at file scope.

With the introduction of function literals, we can declare the struct capture and async_complete (that does not need a name anymore) inside the local scope.

```
   void async(void (*callback)(int result, void* data), void * data);

   int main()
   {
      struct capture {
         int value;
      }* capture = calloc(1, sizeof * capture);

      async((void (int result, void * capture)) {
         struct capture * p = capture;
         free(p);
      }, capture);
   }

   Sample using thrd_create

   int main()
   {
      thrd_t t;

      int r = thrd_create(&t, (int (void * data)) {
         printf("Hello world!\n");
         return 0;
      }, NULL);

      if (r == thrd_success)
```

```
      thrd_join(t, NULL);

   return 0;
}
```

3. SYNTAX AND SEMANTICS

3.1 Syntax

```
   postfix-expression:
      ...
      literal-function

   literal-function:
      ( type-name ) function-body
```

* The type-name shall have a function type.

Note:
The syntax is ambiguous with that of a compound literal. Disambiguation is performed based on the type: only literal functions may have a function type, whereas compound literals shall not.

Note:
Function-specifiers (_Noreturn, inline) and storage-class specifiers (auto, constexpr, extern, register, static, thread_local, typedef) are not permitted in literal function, as their semantics are not currently defined in this context.

3.2 Semantics

* The literal function is a function designator (6.3.3.1)

```
   void main()
   {
      (void (*pf)(void)) = &(void (void)){}; //ok

      //error: lvalue required as left operand of assignment
      &(void (void)){} = 0;
   }
```

* Identifiers from the enclosing scope, excluding labels, are visible inside the literal function.

Samples:

```
   int main() {
      enum E {A};
```

```
      int i = 0;
      (void (void)) {
         int j = sizeof(i); //ok
         enum E e = A; //ok
      }();
   }


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

* For literal functions, __func__ is defined following the same rules as for non-literal functions. The value of __func__ is an implementation-defined null-terminated string when used inside literal functions.

  For comparison, C++ returns "operator ()"
  https://godbolt.org/z/ddesnKzGf

3.3 CONSTRAINS

* The name of an object of automatic storage duration defined in an enclosing function shall be referenced only in discarded expressions.

Samples:

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

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

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

   int main() {
      int f();
      (void ()){ f(); /*ok*/}();
```

```
   }

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

* Variably modified (VM) types defined in an enclosing scope shall not be referenced

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

* An identifier declared outside the body or parameter list of a literal function has the enclosing scope (block or file scope). An identifier declared within the parameter list of a literal function has block scope, which is the literal function body itself. These rules are consistent with those already in effect. See 6.2.1.

  Sample:

```
   int main() {
      (struct X { int i; } (struct Y *y)) {
         struct X x = {};
         return x;
      }(nullptr);

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

4. GENERIC FUNCTIONS

A literal function may result in the instantiation of distinct functions depending on the arguments provided. For example:

```
    #define SWAP(a, b)\
     (void (typeof(a)* arg1, typeof(b)* arg2)) { \
       typeof(a) temp = *arg1; *arg1 = *arg2; *arg2 = temp; \
     }(&(a), &(b))

   int main() {
      int a = 1;
      int b = 2;
      SWAP(a, b);
   }
```

```
#define NEW(...)\
 (typeof((__VA_ARGS__))* (void)){\
    typeof(__VA_ARGS__)* _p = malloc(sizeof * _p);\
      if (_p) *_p = __VA_ARGS__;\
      return _p;\
}()

struct X { const int i; };

int main() {
   auto p2 = NEW((struct X) {});
}
```

5. RATIONALY

Literal functions improve the C language without introducing new concepts, providing more flexibility to functions and enabling a form of generic functions in C.

5.1 Why not C++ lambdas syntax?

Maintaining the existing C grammar is the safest option, as it ensures that function literal syntax stays in sync with function declarations and naturally preserves compatible scope rules

Consider this sample:

```
int main() {
   (struct X { int i; } ()) {
      struct X x = {};
      return x;
   }();
  struct X x = {}; /*ok*/
}
```

The scope rules for the visibility of struct X are the same as those for functions. With the C++ lambda syntax, the return type would be specified after the parameter scope, which could interfere with scope handling or complicate the implementation.

This design also leaves room for alternative capture models that do not follow the C++ approach. Having different models with the same syntax could be confusing for users.

Interoperability using a pair of pointer to function + data is possible in both directions, from C to C++ and from C++ to C, when captureless lambdas are used.

5.2 Why not having captures like C++ lambdas?

When lambdas were introduced in C++, the language already included the necessary infrastructure for capturing such as exceptions, constructors, destructors, and function objects. In contrast, C lacks these features.

Low-level alternatives in C would conflict with existing available patterns, while high-level abstractions might require introducing new concepts that may not fit well in C.

Automatic capture of constexpr objects, or constant objects declared with the register storage-qualifier, from the other scope were considered. This limitation can be removed in the future if necessary. (C++ allows it https://godbolt.org/z/je4vhcecv if you don't take the address)

6. COMPATIBILITY AND IMPACT

* This feature does not break any existing valid C programs, since compound literal objects of type function cannot be created in the current C version.

7. IMPLEMENTATION CONSIDERATIONS

Literal functions can be implemented by the compiler as:

* Anonymous 'static' functions with unique internal names.

* Emission of a unique function symbol in the object file.

7.1 Recommended practice

* Identical literal functions, having the same function type and function-body should, but they are not required, to produce the same function.

7.1 Existing implementations

Cake transpiler has an implementation that converts C2Y code to C99.
http://thradams.com/cake/playground.html

C++ lambda expressions without captures serve as prior art for this feature, albeit with some differences.

8. REFERENCES

* https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3550.pdf
* https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2661.pdf
* https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2924.pdf

9. ACKNOWLEDGEMENTS

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