---------------------- N3679 Function Literals -------------------------

Author : Thiago R Adams
Date   : 2025-09-30
Project: ISO/IEC JTC 1/SC 22/WG 14
Title : Function Literals
Target audience: Implementers, users
Prior art: C++ lambdas without capture

SUMMARY OF CHANGES

* N3679
  - "literal functions" renamed to "function literals"
  - Removed qsort and thrd_create samples
  - Clarifications

* N3645
  - Original proposal

ABSTRACT

This proposal introduces function literals into the C language, providing a syntax for defining functions within expressions. This feature is particularly useful for creating callbacks, which is the primary motivation for the proposal. Its usages also include the ability to create generic functions.

1. 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.

Consider this sample:

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

struct capture { int value; };

static void main_async_complete(int result, void * data) {
   struct capture *capture = data;
   free(capture);
```

```
}

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

Given the current state of the language, the function main_async_complete, which is used only once and is specific to the context in which it is called in main, must be declared at file scope.

Since it uses the struct capture, which is also tied to that specific callback in that particular context, the struct must likewise be declared at file scope.

With the introduction of function literals, we can declare the struct capture and main_async_complete (which no longer needs a name) inside the local scope, keeping all these tightly related parts together.

The syntax for function literals is similar to that of compound literals, with the difference that the type is a function type, and instead of an initializer list, we have the function body.

```
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);
}
```

The cast from void * to struct capture is much safer, since we can see the correspondence with the object passed as an argument.

I believe this correspondence can be improved with future proposals defining this relationship in the type system or through attributes. It will not be defined here, but this sample provides a glimpse of the idea.

```
void async(void (*callback)(int result, [[type(T)]] void* data),
           [[type(T)]] void * data);

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

   int dummy;
   async((void (int result, void * capture)) {

      /* warning: capture is pointing to an object of type 'int' */
      struct capture *p = capture;

      free(p);
   }, &dummy);
}
```

2. SYNTAX AND SEMANTICS

2.1 Syntax

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

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

The syntax is ambiguous with that of a compound literal. Disambiguation is based on the type: function literals have a function type, whereas compound literals do not.

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

2.2 Semantics

The function literal is a function designator.

Particularly, taking the address of a function literal returns the address of the function (not the address of a pointer), and a function literal is not an lvalue.

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

   &(void (void)){} = 0;  /* error: lvalue required */
}
```

Function literals 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 function literal.

```
int main() {
   void f();
   enum E {A};
   (enum E (enum E arg)) {
      enum E e = A;
      f();
      return e;
   }(A);
}
```

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

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

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

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

Objects with automatic storage, declared in the enclosing scope and which are not VM types, can be used within the return type, arguments, and function body of function literals, provided they are used in expressions discarded at some point within the function literal.

Samples:

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

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

int main() {
  int i;
  1 || (int (void)) { return i; /* 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 function literal.

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

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

The value of __func__ is an implementation-defined null-terminated string when used inside function literals. For comparison, C++ lambdas return "operator ()".

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

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

Sample:

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

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

3. GENERIC FUNCTIONS

A function literal may be used in function-like macros, allowing a form of generic functions in C.

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);

   double da = 1.0;
   double  db = 2.0;
   SWAP(da, db);
}
```

Distinct function literals are not required to have unique addresses.

```
int main(){
   auto pf1 = (void ()) { return 1 + 1; };
   auto pf2 = (void ()) { return 2; };
```

```
    auto pf3 = (void ()) { return 2; };
    /* pf1 and pf2 and pf3 can have the same address */
}
```

This allows implementations to reuse the same function literal, which is important to avoid code bloat caused by function literals inside function-like macros.

Note: When static objects are used inside function literals, they have unique addresses. Thus, in this case, the function literal will also be unique.

```
int main() {
    auto pf1 = (void ()) { static int i = 0; };
    auto pf2 = (void ()) { static int i = 0; };
    assert(pf1 != pf2);
}
```

4. RATIONALE

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

4.1 Why not C++ lambda syntax?

Maintaining the existing C grammar is the safest option, as it ensures that function literal syntax always stays in sync with function declarations and naturally preserves existing scope rules, making the C standard and language more concise.

Consider this sample:

```
int main() {
   (struct X * (struct X * p)) {
      return p;
   }(0);
}
```

The tag X, declared at the return type, can be used in the parameters. With the C++ lambda syntax, the return type would be specified after the parameter, which could interfere with scope rules.

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.

4.2 Why not have 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.

Capturing constexpr objects or constants declared with the register storage qualifier from the enclosing scope was considered.Although this limitation might be lifted in the future, the workaround is simply to use constant objects with static storage duration.

Note: For comparison, C++ lambdas without captures can use constexpr objects, provided their addresses are not taken.

5. 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.

6. EXISTING IMPLEMENTATIONS

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

A combination of two GCC extensions—statement expressions and nested functions—gives us something similar to function literals.
For instance:

```
int main() {
   ({int _(int a) { return a * 2; } _;})(2);
}
```

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

7. REFERENCES

1 N3645 - Literal functions
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3645.pdf

2 N2924 - Type-generic lambdas
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2924.pdf

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

4 N3678 - Local functions
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3678.pdf

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

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

8. WORDING

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

9. ACKNOWLEDGEMENTS

I would like to recognize the following people for their help in this work: Joseph Myers, Martin Uecker, Jens Gustedt, Javier Múgica, Alejandro Colomar, Christopher Bazley.