Local functions
&&
Function literals

N3678 and N3679

Motivation

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

struct start_capture {
   int value;
};

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

void start() {
   struct start_capture* capture = calloc(1, sizeof *capture);
   async(capture, start_callback);
}
```

do this, then that...

```
void part1_async(void* data, void (*callback)(int result, void* data));
void part2_async(void* data, void (*callback)(int result, void* data));

struct part1_capture {
   int value;
};

struct part2_capture {
   char ch;
};

static void part2_complete(int result, void* data);
```

```
static void part1_complete(int result, void* data) {
   struct part1_capture* capture1 = data;

   struct part2_capture* capture2 = calloc(1, sizeof *capture2);
   part2_async(capture2, part2_complete);

   free(capture2);
}

static void part2_complete(int result, void* data) {
   struct part2_capture* capture2 = data;
   free(capture2);
}

void start() {
   struct part1_capture* capture1 = calloc(1, sizeof *capture1);
   part1_async(capture1, part1_complete);
}
```

Motivation

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

struct start_capture {
   int value;
};

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

void start() {
   struct start_capture* capture = calloc(1, sizeof *capture);
   async(capture, start_callback);
}
```

Local functions

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

```
void start() {

   struct capture {
      int value;
   };

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

   struct capture* capture = calloc(1, sizeof * capture);
   async(capture, callback);
}
```

Literal functions

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

void start() {

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

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

Reusing captures

```
int main()
{
    struct capture { int id; }* capture = calloc(1, sizeof * capture);
    login_async(capture,  (void (int id, void * p))
    {
        printf("login completed. id=%d\n", id);
        struct capture * cap1 = p;
        cap1->id = id;
```

```
        get_data_async(cap1 /*moved*/, (void (const char* email, void * data))
        {
            struct capture * cap2 = data;
            printf("your data='%s'  from id=%d\n", email, cap2->id);
            free(cap2);
        });
    });
}
```

Captures on Stack

```
#include <stdio.h>

void for_each_file(const char* path,
                   void* data,
                   void (*f)(void* data, const char* filename)){}

int main(){

   struct {
      enum { ALL, SMALL } filter;
      bool more_data;
    } capture = { ALL, false };

   for_each_file("c:", &capture, (void (void* p, const char* file_name))
   {
     typeof(capture) * captured = p;
     if (captured->filter == ALL) {}
   });
}
```

Static captures

```
#include <stdio.h>

void for_each_file(const char* path,
                   void* data,
                   void (*f)(void* data, const char* filename));

int main()
{
   static int filter = 1;
```

```
   for_each_file("c:", 0, (void (void* data, const char* file_name))
   {
       if (filter == 1) {}
   });
}
```

Async implementation I

```
void async(void* data, void (*callback)(int result, void* data))
{
  struct capture {
      void * data;
      void (*callback)(int result, void* data);
   } capture = calloc(1, sizeof * capture);

   if (capture == NULL) {
      callback(1, data);
      return;
   }

   capture->data = data;
   capture->callback = callback;

   thread_pool(capture, (void (void* data))
   {
      struct capture* capture = data;
      capture->callback(0, capture->data);
      free(capture);
   });
}
```

Async implementation II

```
void async(void* data, void (*callback)(int result, void* data))
{
  struct capture {
      void * data*;
      void (*callback)(int result, void* data);
   } capture = {data, callback};

   thread_pool(capture, sizeof capture, (void (void* data))
   {
```

```
      struct capture* capture = data;

      /* task */

      capture->callback(0, capture->data);
   });
}
```

Async implementation III

```
void async(void* data, void (*callback)(int result, void* data))
{
  struct capture {
      void * data*;
      void (*callback)(int result, void* data);
   } capture = {data, callback};

   thread_pool(capture, sizeof capture, (void (void* data))
   {
      struct capture* capture = data;

      /* task */

      dispatch(capture, sizeof capture, (void (void* data)))
      {
         struct capture* capture = data;
         capture->callback(0, capture->data);
      });
   });
}
```

Local function syntax

```
block-item:
    ...
    <span style="color:blue">function-definition</span>

function-definition:
   attribute-specifier-sequence opt declaration-specifiers declarator function-
body
```

Forward declarations

```
#include <stdio.h>

int main() {
   void f();
   f();
   void f() { printf("local"); }
}

void f() { printf("extern"); }
```

GCC Nested function solution

```
#include <stdio.h>

int main() {
   auto void f();
   f();
   /*auto*/ void f() { printf("local"); }
}

void f() { printf("extern"); }
```

> See also: N3579 auto as a placeholder type specifier

GCC Nested function solution

```
#include <stdio.h>

int main() {
   void f();
   f();
   /*error: static declaration of 'f' follows non-static declaration*/
   void f() { printf("local"); }
}

void f() { printf("extern"); }
```

Alternative I (static)

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

int main() {

   /*local functions*/
   static void f(); /*local function declaration*/
   static void f() { }

   /*GCC nested function*/
   auto int f2();
   int i = 1;
   int f2() { return i; }
}
```

Alternative II (hybrid)

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

int main() {
   static void f();
   void f() { }

   /*static*/ int f1() { return 0; }

   int i = 1;
   int f1() { return i; /*GCC extension*/ }
}
```

Alternative III (same as gcc)

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

int main() {
   auto void f();
   /*auto*/ void f() { }

   auto int f2();
   int i = 1;
```

```
   int f2() { return i; /*GCC extension*/ }
}
```

GCC Curiosity I

```
#include <stdio.h>

int main()
{
    auto void local();

    // error: nested function 'local' declared but never defined
    void local() {
        printf("1 ");
    }

    local();

    auto void local();

}
```

GCC Curiosity II

```
#include <stdio.h>

int main()
{
    auto void local();

    void local() { printf("1 "); }

    local(); //prints 2

    auto void local();

    void local() { printf("2 "); }
    local(); //prints 2
}
```

Local functions and scope

- A local function must have only one definition per scope
- Forward declarations are scoped.
- warning: Forward declaration after the definition
- warning: Multiple forward declarations

> obs: not at the N3678 yet

Function Literal syntax

```
postfix-expression:
   ...
   <span style="color:blue">function-literal-definition</span>

<span style="color:blue">function-literal-definition:
   ( attribute-specifier-sequence opt declaration-specifiers abstract-declarator
)
         function-body</span>

function-definition:
   attribute-specifier-sequence opt declaration-specifiers declarator function-
body
```

> The abstract-declarator portion of a function literal definition must have a function type.

> Disambiguation: Compound literals cannot have a function type.

Why not C++ lambda syntax?

- Keeps the grammar for functions and function literals in sync.
- Keeps the existing scope rules for return types and parameters.
- Return type deduction not required. (could be added with auto)

```
/*   C++   */
int main() {
    //error: return type 'struct main()::X' is incomplete
    [] () ->  struct X { int i; } * {
```

```
        return 0;
    }();
}
```

> This could be necessary for someone using `vec(int)` with struct tag compatibility.

Why not C++ lambda syntax?

- Fits well with the existing concept of compound literals.
- Do not create the expectation that the C and C++ features are identical.

```
/*   C   */
int main() {
    (struct X { int i; } *(void)) {
        return 0;
    }();
    struct X x;
}
```

> It does not create problems that were not considered before.

Semantics

- The function literal is a function designator. (Behaves like a function, not a function pointer)

```
void main()
{
    (void (*pf1)(void)) = (void (void)){}; /* ok */
    (void (*pf2)(void)) = &(void (void)){}; /* ok */
    &(void (void)){} = 0;                   /* error: lvalue required */
}
```

File scope function literals

```
auto f = (int (int a)){ return a * 2; }; /* ok */

int main()
```

```
{
}
```

> I don't have a use case for that at the moment.

Labels

- Labels are not shared
- Statements are not shared (break; continue)

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

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

Returning VM types

```
#include <stdio.h>
int main() {
    int n = 1;
    auto typeof(int [n])* local(void);

    n = 2;
    typeof(int [n])* local(void) {
       return 0;
    }

    n = 3;
    auto r = local();

    n = 4;
    printf("%zu", _Countof(*r)); //returns 2
}
```

> Following GCC implementation of nested functions

Argument evaluation

```
#include <stdio.h>

int main() {
    int n = 1;

    void local(typeof(int[n])* p)
    {
         //does it need address of n?
         printf("%zu", _Countof(*p));
    }
    n = 2;
    int a[n];
    local(&a);

    //void (*pf) (typeof(int[n])* p) = local;
    //pf(&a);
}
```

Argument evaluation

```
#include <stdio.h>

int main() {
    int n = 1;

    void local(int n, typeof(int [n])* a)
    {
         printf("%zu", _Countof(*a)); //prints 2
    }

    int a[2] = {1, 2};
    local(2, &a);
}
```

> Following GCC implementation of nested functions

func

- The value of the string returned by **func** is implementation-defined.

> GCC returns the function name for nested functions

> C++ returns "operator ()" in lambdas

Scope

- Function literals and local functions have access to the enclosing scope at the point of its definition.

```
int main() {

    struct X {int i; };
    enum E {A};

    (void (void)) {
        struct X x = {}; /* ok */
        x.i = A;         /* ok */
    }();

   void local() {
     struct X x = {}; /* ok */
     x.i = A;         /* ok */
   };
}
```

Automatic variables

- Identifiers referring to automatic variables of an enclosing function cannot have their address resolved inside the body of a function literal or local function. If they have VM types, this restriction also apply to resolving their type.

```
int main() {
    int i = 2;
    void local() {
        int j = sizeof(i); /* ok */
        int  k = i;        /* constraint violation */
        int *p = &i;   /* constraint violation */
```

```
    };
}
```

Automatic variables

- Identifiers referring to automatic variables of an enclosing function cannot have their address resolved inside the body of a function literal or local function. If they have VM types, this restriction also apply to resolving their type.

```
void start(int n) {
    int a[n];
    void local() {
        typeof(a) k;       /* constrain violation */
        int m = sizeof(a); /* constrain violation */
    };
}
```

Constants

- The same restrictions apply to constants. However their values can be read without accessing memory.

```
int main() {
   constexpr int a = 1;
   const int b = 2;     /*N3693 Implicitly constexpr*/
   const int j = get();

   void local() {
     int x = a;   /* ok */
     int *p = &a; /*constrain violation*/

     x = b;       /* ok */
     p = &b;      /*constrain violation*/

     x = j;       /*constrain violation*/
   };
}
```

> Originally left as a possible option, it can be included

Non-automatic variables

```
static int g = 1;

int main() {
   static int i = 1;

   void local() {
     int j = sizeof(i); /* ok */
     int k = i;         /* ok */
     int m = g;         /* ok */
   };
}
```

Generic functions

```
#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);

    (void (typeof(a)* arg1, typeof(b)* arg2)) {
        typeof(a) temp = *arg1;
        *arg1 = *arg2;
        *arg2 = temp;
    }(&(a), &(b));


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

Function Literal address

- Distinct function literals are not required to have unique addresses.
- Extend this to local functions?

```
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 */
}
```

Static variables inside function literals

- static variables inside function literals will generate distinct functions

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

Key points

- Almost zero learning curve
- Existing practice
- Does not require trampolines or other hidden features.
- If it looks like a function, then it is a function.
- No forced capture strategy (by reference, by copy, stack, heap, etc.).
- Works with existing APIs that use `void *` callbacks
- We are not adding new problems.
- We are adding convenience improving safety and maintainability.
- Keeps the compiler simple

Road map

- Improving the proposal, add wording maybe merge in one proposal?
- Deciding on forward-declaration syntax (static x auto)

- Experimental implementation http://cakecc.org/ (missing VM types)

References

- N3724: Discarded
- N3622 Allow calling static inline within extern inline
- N3579: auto as a placeholder type specifier
- N3693: Integer Constant Expression
- N3694: Functions with Data
- N3654: Accessing the Context of Nested Functions
- Reddit: https://www.reddit.com/r/C_Programming/comments/1omrrra/closures_in_c_yes/
- http://cakecc.org/

Thank You

Press ← → or Space to navigate.

Function literal emulation in GCC

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

Multiple forward declarations/definitions

```
#include <stdio.h>

int main()
{
    int n = 1;
    auto typeof(int [n])* f();
    printf("%zu\n", _Countof(*f())); //1

    n = 2;
    auto typeof(int [n])* f();
```

```
    printf("%zu\n", _Countof(*f())); //2

    n = 3;
    auto typeof(int [n])* f(){
    }
    printf("%zu\n", _Countof(*f())); //3
}
```