Defer Mechanism for C

Robert C. Seacord

nccgroup

Agenda

Compiler Optimizations

Constant Folding

Adding a Pointer and an Integer

Integer Overflow

GCC Options

Strict Aliasing

Pointer Provenance

Optimization Suggestions

C11 Analyzability Annex

Summary and Recommendations

Defer Mechanism

The defer mechanism can restore a previously known property or invariant that is altered during the processing of a code block.

- useful for paired operations, where one operation is performed at the start of a code block and the paired operation is performed before exiting the block.
- pattern is common in
  - resource management
  - Synchronization
  - outputting balanced strings (e.g., parenthesis or HTML).

Resource Management

Examples of C standard library functions that acquire resources include:

- allocated storage: **malloc**, **calloc**, **realloc**, **aligned_alloc**, **strdup**, **strndup**
- streams: **fopen**, **freopen**
- temporary file: **tmpfile**
- threads: **thrd_create**
- thread specific storage: **tss_create**
- condition variable: **cnd_init**
- condition variable: **cnd_wait**
- mutexes: **mtx_init**, **mtx_lock**, **mtx_timedlock**, **mtx_trylock**

Differing Behaviors

Resource	Can Fail	Reports Failure	Released on Thread Exit	Released on Program Exit
Allocated storage: **`free`**	✓	✗	✗	✓
Thread-specific storage key: **`tss_delete`**	✓	✓	✗	✓
Thread-specific storage: destructor	✓	✗	✓	✓
File pointer: **`fclose`**	✓	✓	✗	✓
File: **`remove`**	✓	✓	✗	✗

Security Concerns

A *denial-of-service* (DoS) attack occurs when legitimate users are unable to access information systems, devices, or other network resources resulting from the actions of an adversary.

DoS attacks attempts frequently take the form of a resource-exhaustion attack that makes a computer resource unavailable or insufficiently available to the application.

Double Free vulnerabilities can be exploited to execute arbitrary code with the permissions of a vulnerable process.

A common source of this error are developers who deallocate memory while handling an error condition but then deallocate it again during normal cleanup procedures.

Defer Statement

A **defer** statement defers the execution of a *deferred statement* until the containing guarded block terminates

Deferred statement is sequenced in last-in-first out (LIFO) order after all statements just before the guarded block terminates.

The defer statement may be introduced into the grammar as follows:

statement:

attribute-specifier-sequence$_{opt}$ *defer-statement*

defer-statement:

defer *statement*

A block can contain multiple **defer** statements.

Stack Unwinding

Before the normal processing of the termination event, the C library functions **exit** and **thrd_exit** trigger an execution of all deferred statements for the current thread by unwinding the stack.

Deferred statements shall not

- include **return** statements.
- call functions that may result in termination of the current thread or the whole program execution other than by calling the **panic** or **abort** functions.
- contain a **goto** or **longjmp** that targets a location outside the deferred statement
- contain a label or call to **setjmp** that are the target of a **goto** statement or **longjmp** call, respectively, outside the deferred statement.

Should defer statements be static or dynamic?

How many times and in what exact order should deferred statements be executed?
The dynamic approach matches programmer expectations based on control flow:

```
{
  if (x) defer whatever(x);
  int i;
  for (i = 0; i < n; i++) {
    printf("up: %d\n", i);
    defer printf("down: %d\n", i--);
  }
}
```

deferred only if **x** evaluates to a non-zero value at runtime

Pushed for each iteration of the loop in which the **defer** statements are encountered

nccgroup

Dynamic Approach

Resources are allocated at runtime

- additional runtime overhead
- operations might fail

Loops may also be constructed with **goto** statements and labels, as long as they do not exit the deferred or guarded block

Static approach

Statically allocate resources at compile time

- eliminate the possibility that deferred statements may fail at runtime

The compiler provides a slot for each deferred statement in the same way that it would for an unconditional defer.

A **defer** statement records if it has been triggered or not.

If reached, the deferred statements are executed once at the end of the corresponding block.

If the loop is not entered, the deferred statement is not executed.

deferred statements are executed in reverse lexical order.

Should object values be captured?

Identifiers accessed by the deferred statement must be visible within the **defer** statement and its scope extended to at least the end of the guarded block.

Should object values accessed in deferred statements should be captured when the deferred statement is encountered or their latest values read when the deferred statements are evaluated?

Lambdas with copy and reference semantics would be a great solution that is not yet available for C.

Last Value

```
{
  char *ptr = malloc(SZ);
  defer free(ptr);
  // ...
  ptr = realloc(ptr, SZ * 2);
  // ...
}
```

The latest value stored in `ptr` is passed to `free`.

Captured Values

```
{
  struct s *ptr = malloc(sizeof(struct s) * 10);
  defer free(ptr);
  for (int i = 0; i < 10; ++i) f(ptr++);
}
```

Assumes `ptr` is captured.

Capture pairs better with the dynamic approach.

```
if (function_which_sets_errno())
  defer printf("%d", errno);
```

Assumes `errno` is captured.

User Expectations “study” / Twitter Poll

If C added "deferred statements" that execute just before the block exits, what would you expect the output of this code block to be?

```
{
  int i = 0;
  defer printf("%d", ++i);
  i = 12;
}
```

C++ programmers more likely to assume captured values

The results from 387 responses show a 2:1 preference for the value being read at the time the deferred statements are executed (66.9%) rather than when the defer statement encountered (33.1%).

Do we want a **guard** keyword?

An explicit guard keyword allows the following code to be written:

```
guard {
  void *ptr = malloc(12);
  if (ptr) {
    defer free(ptr);
    // Use ptr
  }
  // Use ptr some more
} // free ptr here
```

All Scopes Guarded

The previous example can be rewritten without the guard keyword as follows:

```
{
  void *ptr = malloc(12);
  defer if (ptr) free(ptr);
  if (ptr) {
    // Use ptr
  }
  // Use ptr some more
} // conditional free ptr here
```

Comparison

Making the guarded block explicit

- is strictly more expressive
- allows the programmer more control over when the deferred statements are evaluated.

The explicit syntax adds an additional keyword.

Guarding all scopes

- provides a more terse syntax
- eliminates any issues with introducing a new keyword
- familiar for C++ programmers

Panic and recover

The primary purpose of defer is to manage the release of resources.

The primary purpose of a panic/recover mechanism is error handling.

Panic/recover depend on the defer mechanism to release resources, but defer is separable from panic/recover.

Panic/recover are similar to **throw**/**catch** in C++ while **defer** is similar to RAII.

Panic

A panic may potentially be the result of a trap, such as an invalid arithmetic operation or the result of invoking either of the following forms of the panic macro:

```
#include <stddefer.h>
typedef int (*panic_handler_t)(int);
_Noreturn void panic(int code);
_Noreturn void panic(int code, panic_handler_t handler);
```

The **panic** macro indicates an abnormal execution condition and will *unwind* the caller's stack (see Appendix I) and execute all deferred statements registered in that stack frame.

The **recover** function

Once execution starts inside a deferred statement, the condition that leads there can be investigated by invoking the **recover** function.

The **recover** function returns an integer value that indicates the reason the deferred statement is executing.

If the return value is equal to zero, the execution of the deferred statement is the result of the regular termination of the guarded block caused by reaching the **}** that terminates the block, execution of a **break** or **return** statement, the invocation of the **exit** or **thrd_exit** functions, or by a call to **panic** with a zero value.

In that case, processing of deferred statements continues as if the **recover** function had not been called.

The **recover** function

If the **recover** function returns a value other than zero, the thread or program is panicking.

In this case, processing of deferred statements stops with the termination of the current deferred statement.

Once a non-zero error condition has been recovered, the responsibility for the condition is passed to the application.

A new panic can be triggered by calling either form of the **panic** macro.

Recover Example 1

```
void g(int i) {
  if (i > 3) {
    puts("Panicking!");
    panic(i);
  }
  guard {
    defer {
      printf("Defer in g = %d.\n", i);
    }
    printf("Printing in g = %d.\n", i);
    g(i+1);
  }
}
```

Recover Example 2

```
void f() {
  guard {
    defer {
      puts("In defer in f");
      fflush(stdout);
      int err = recover();
      if (err != 0) {
        printf("Recovered in f = %d\n", err);
        fflush(stdout);
      }
    }
    puts("Calling g.");
    g(0);
    puts("Returned normally from g.");
  }
}
```

Output

```
Calling g.
Printing in g = 0.
Printing in g = 1.
Printing in g = 2.
Printing in g = 3.
Panicking!
Defer in g = 3.
Defer in g = 2.
Defer in g = 1.
Defer in g = 0.
In defer in f
Recovered in f = 4
Returned normally from f.
```

Questions

Do we want a defer statement?

Should defer statements be static or dynamic?

Should object values be captured?

Should it be determined by scope alone?

Do we want a panic/recover mechanism?

25