C language mechanism for error handling and deferred cleanup

Jens Gustedt, Robert C Seacord

C language mechanism for error handling and deferred cleanup

extended abstract

Jens Gustedt
INRIA and Université de Strasbourg
France
jens.gustedt@inria.fr

Robert C. Seacord
NCC Group
USA
robert.seacord@nccgroup.com

ABSTRACT

This paper introduces the implementation of a C language mechanism for error handling and deferred cleanup adapted from similar features in the Go programming language. This mechanism improves the proximity, visibility, maintainability, robustness, and security of cleanup and error handling over existing language features. This feature is under consideration for inclusion in the C Standard. The library implementation of the features described by this paper is publicly available under an Open Source License at https://gustedt.gitlabpages.inria.fr/defer/.

ACM Reference Format:
Jens Gustedt and Robert C. Seacord. 2021. C language mechanism for error handling and deferred cleanup: extended abstract. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3412841.3442116

1 INTRODUCTION

The defer mechanism can restore a previously known property or invariant that is altered during the processing of a code block. The defer mechanism is 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. Because blocks can be exited using a variety of mechanisms, operations are frequently paired incorrectly. The defer mechanism in C is intended to help ensure the proper pairing of these operations. This pattern is common in resource management, synchronization, and outputting balanced strings (e.g., parentheses or HTML tags). A panic/recover mechanism allows error handling at a distance.

Unlike existing high-level languages, C lacks a general mechanism for resource management and error handling, outside of C's use of integer error codes and outmoded mechanisms such as **errno**. Consequently, resource management in C programs can be complex and error prone, particularly when a program acquires multiple resources. Each acquisition can fail, and resources must be released to prevent leaking. If the first resource acquisition fails, no cleanup is needed, because no resources have been allocated. However, if the second resource cannot be acquired, the first resource needs to be released. Similarly, if the third resource cannot be acquired, the second and first resources need to be released, and so forth. This pattern results in duplicate cleanup code, and it can be error-prone because of this duplication and the associated complexity.

C programmers need to manage the acquisition and release of resources. Because resources exist in limited quantities, it is always possible that a resource cannot be acquired because the supply of that resource has been exhausted. Examples of C standard library functions [2] that acquire resources include storage functions (**malloc**, **calloc**, **realloc**, **aligned_alloc**), mutexes (**mtx_init**, **mtx_lock**, **mtx_timedlock**, etc.), strings (**strdup**, **strndup**), and streams (**fopen**, **freopen**).

Improper resource management and error handling frequently results in software vulnerabilities. They can result in denial-of-service (DoS) attacks which seek to prevent legitimate users from being able to access information systems, devices, or other network resources [3]. For example, if an attacker can identify an external action that causes memory to be allocated but not freed, memory can eventually be exhausted. Once memory is exhausted, additional allocations fail, and the application is unable to process valid user requests. Another variant of such erroneous resource management can be exploited in multi-threaded programs that use mutexes. By default, most systems have no provisions to cope with a mutex that is locked by a thread that exits. Other threads that try to access that same mutex will block, eventually causing a deadlock of the whole execution.

Another common error associated with manual memory management is the deallocation of memory more than once, without an intervening allocation. *Double Free* vulnerabilities can be exploited to execute arbitrary code with the permissions of a vulnerable process [4]. A common source of this error is the deallocation of memory while handling an error condition which is then deallocated again during normal cleanup procedures.

To cope with these shortcomings, the C standards committee (ISO/IEC JTC1/SC22/WG14) is investigating the adaptation of Go's mechanism of deferred execution to C [1]. There is evidence that this mechanism provides improved cleanup and error handling over existing C language features (such as **goto** or **longjmp**) and over features that could be imported from other languages such as C++, C# or Java like constructor/destructor pairings, exception handling, or finally blocks, or from compiler extensions such as explicit destructor functions (GCC compilers).

This paper provides a concise overview of the features that are proposed by our reference implementation.

2 PRINCIPAL FEATURES

The principal features of the proposed language mechanism are:

- **guard** prefixes a guarded block
- **defer** prefixes a defer clause

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8104-8/21/03.
https://doi.org/10.1145/3412841.3442116

- **break** ends a guarded block and executes all its defer clauses
- **return** unwinds all guarded blocks of the current function and returns to the caller
- **thrd_exit** unwinds all active deferred statements of the current thread and exits that thread
- **exit** unwinds all active deferred statements of the current thread and exits the execution
- **panic** starts global unwinding of all guarded blocks
- **recover** stops a panic and provides an error code

The existing features **break**, **return**, **thrd_exit** and **exit** gain new functionality while the remaining features are new. Existing features are overloaded by macro definitions that take effect as soon as the library header is included.

2.1 The defer Statement

The **defer** statement defers execution of the *deferred statement* to the end of the guarded block (described in the next subsection). The syntax is

defer *statement*

Deferred statements may not themselves contain guarded blocks or other defer statements, and must not call functions that may result in a termination of the execution other than **panic**. Additionally, such a call to **panic** must only occur **after** the current panic state has been tested with **recover**.

The deferred statement may use any local variable that is visible where the **defer** is placed and that is alive when the deferred statement is executed at the end of the encompassing **guard** or function body. This property is verifiable at compile time, and a violation usually results in compilation failure. The reference implementation puts everything in place such that a deferred statement uses the last-written value for all variables.

2.2 The guard Statement

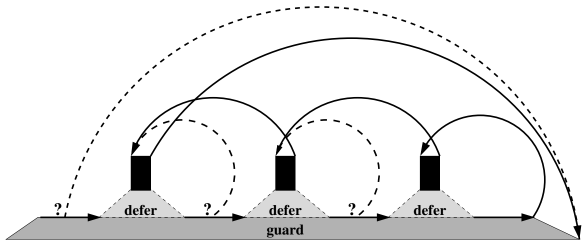
The **guard** statement marks a whole block as guarded for uses of the defer mechanism.

guard *compound-statement*

If such a block terminates normally or with a **break** or **continue** statement, all deferred statements that have been registered are executed in reverse order of their registration. There is also a macro **defer_break** that can be used in contexts where **break** or **continue** statements would refer to an inner loop or **switch** statement. In addition, **return**, **exit**, **quick_exit**, and **thrd_exit** also trigger the execution of deferred statements, but continue this treatment up to their respective levels of nesting of guarded blocks. In contrast, other standard means of non-linear control flow out of, or into, the block (**goto**, **longjmp**, **_Exit**, and **abort**), do not invoke that mechanism and may result in memory leaks or other damage when used within such a guarded block.

In the C standards committee proposal [1], each function body implements a guarded block such that an explicit **guard** statement is not necessary for many common use cases. We use the term "guarded block" throughout this paper, regardless of whether a **guard** statement is used or the function body defines the guarded block. Listing 1 shows a guarded block containing three deferred statements. The **defer** keyword indicates that the evaluation of the following statement, such as a call to **free**, is deferred to the end of the guarded block. The deferred statement is evaluated regardless of how the guarded block exits. In Listing 1, the block can be exited after the final statement is evaluated, or if a **break** statement is evaluated. Deferred statements are evaluated in the inverse order they were encountered. Possible branches for this guarded block are shown in Figure 1. Dashed lines represent conditional error handling paths that are executed when resources are unavailable or evaluation is interrupted by a signal.

Figure 1: Control flow of a guarded block with three defers



Listing 1: An example with three deferred statements

```
guard {
  void * const p = malloc(25);
  if (!p) break;
  defer free(p);
  void * const q = malloc(25);
  if (!q) break;
  defer free(q);
  if (mtx_lock(&mut)==thrd_error) break;
  defer mtx_unlock(&mut);
  // all resources acquired
}
```

This approach has advantages over familiar C, C++, C# or Java solutions. Cleanup code that releases resources is collocated with the code that acquires these resources, making it easier for programmers to ensure statements are properly paired. Similar to C# or Java's finally blocks, cleanup code is clearly visible. This differs from **atexit** handlers in C or constructor/destructor pairs in C++, where cleanup code may be defined in a different translation unit.

Listing 2: Emulation of defer by goto

```
   void * const p = malloc(25);
   if (!p) goto DEFER0;
   if (false) {
     DEFER1: free(p); goto DEFER0;
   }
   void * const q = malloc(25);
   if (!q) goto DEFER1;
   if (false) {
     DEFER2: free(q); goto DEFER1;
   }
   if (mtx_lock(&mut)==thrd_error) goto DEFER2;
   if (false) {
     DEFER3: mtx_unlock(&mut); goto DEFER2;
   }
   // all resources acquired
   goto DEFER3;
   DEFER0:;
```

For control flow that doesn't include **return**, **exit**, or other non-returning functions calls an equivalent control flow can be implemented with existing C language features. The guarded statement from Listing 1 is equivalent to the code segment in Listing 2. The **if (false)** statement guarantees that the deferred statements are not evaluated when they are first encountered. The labels and **goto** statements implement the backward branches to execute the deferred statements when the guarded block terminates.

Listing 3: A linearization

```
    void * const p = malloc(25);
    if (!p) goto DEFER0;
    void * const q = malloc(25);
    if (!q) goto DEFER1;
    if (mtx_lock(&mut)==thrd_error) goto DEFER2;
    // all resources acquired
    mtx_unlock(&mut);
  DEFER2: free(q);
  DEFER1: free(p);
  DEFER0:;
```

A common C idiom for cleanup handling is to *linearize* resource management as shown in Listing 3. This code has the advantage of making the conditional error handling code explicit but at the cost of proximity; the cleanup code is removed from where its need arises. This linearization requires a naming convention for the labels. For more complicated code the maintenance of these jumps can be error prone.

In contrast to such commonly found idioms, deferred statements do not depend on arbitrary label names (C) or RAII classes (C++) and do not change when **defer** or **break** statements are added or removed.

All exits from a guarded block by **break**, **return**, **thrd_exit**, **exit**, or an interruption by a signal must be detected and acted upon. This is difficult to implement in C and requires **try/catch** blocks in C++. The defer mechanism ensures that deferred statements are guaranteed to be executed, provided program execution progresses. The **defer** statement has syntax similar to other control structures, such as **if** and **for** blocks.

For maintainability and robustness we opted for object access by reference within a deferred statement and not by value. Listing 1 illustrates the importance of access by reference. The value of p could be changed by means of **realloc**, for example. If **defer** copies the value, the deferred call to **free** would use a stale copy of the value.

Because the scope in which the deferred statement is executed is limited, compilers can detect and diagnose a usage of a variable that is out of scope. As stated earlier, vulnerabilities can occur if resources are not released, or released more than once. Security is improved because of increased proximity and visibility of cleanup code to the resource acquisition code, and to improvements in maintainability and robustness.

2.3 Regular termination

There are three severity levels for termination of a guarded block: **break** or similar just terminates the guarded block, **return** terminates all guarded blocks of the same function and **exit** or **panic** terminates all guarded blocks in the same thread across all function calls. The implementation augments **break** (within a **guard**) and **return** by this functionality. This is achieved by overloading these features by macros, a technique that the C standard explicitly allows. Code that is compiled with the <stddefer.h> header will also replace calls to the C library functions **exit**, **thrd_exit**, and **quick_exit** by calls to their corresponding wrappers.

2.4 Panic/Recover

Error handling at distance is supported by the introduction of a panic/recover mechanism, which is similar to throw/catch in C++. Panic/recover depends on the defer mechanism to release resources during stack unwinding. Generally, a *panic* is a condition which requires unwinding one or several levels of deferred statements. Such a panic may occur implicitly if the run-time library detects or signals a fault, or explicitly by the use of a macro dedicated to that purpose.

The **panic** macro is called to indicate an abnormal execution condition. It triggers the execution of all active deferred statements of the current thread in the reverse order they are encountered, until either a deferred call to **recover** is executed or all deferred statements have been executed. If no recover statement is encountered, the function stack unwinds the caller's stack and executes all deferred statements registered in that stack frame. This process continues until a recover expression is encountered or all deferred statements have executed.

Our reference implementation applies the convention that user error codes passed to **panic** are always positive, and system error codes are generally negated **errno** numbers.

Once a deferred statement begins evaluation, the condition that led there can be investigated by means of **recover**. The **recover** function returns an integer value indicating why the deferred statement is executing. A return value of 0 indicates that the deferred statement is the result of a **break**, **return** or **exit**. Any other value indicates that the deferred statement is executing as the result of a **panic**. The programmer is responsible for handling the recovered error and may re-issue the panic from within the deferred statement if it is impossible to recover at a particular level of abstraction.

3 CONCLUSIONS

An effort is underway to add a mechanism for error handling and deferred cleanup to the C standard by providing a new library clause for a new header with the proposed name <stddefer.h> [1].

C code converted to use defer typically has less code, equal or better performance, and improved usability based on an analysis of existing open source libraries such as OpenSSL, PostgreSQL, and Flux.

ACKNOWLEDGMENTS

Thanks to Aaron Ballman, Alex Gilding, JeanHeyd Meneide, Miguel Ojeda, Tom Scogland, Martin Uecker, Freek Wiedijk and Simon Harraghy for their contributions to this work.

REFERENCES

[1] Aaron Ballman, Alex Gilding, Jens Gustedt, Tom Scogland, Robert C. Seacord, Martin Uecker, and Freek Wiedijk. 2020. Defer Mechanism for C. ISO SC22 WG14 N2542, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2542.pdf.
[2] ISO. 2018. *ISO/IEC 9899:2018: Programming languages — C.* International Organization for Standardization, Geneva, Switzerland.
[3] Jelena Mirkovic, Sven Dietrich, David Dittrich, and Peter Reiher. 2004. *Internet Denial of Service: Attack and Defense Mechanisms.* Prentice Hall, Upper Saddle River, NJ, USA. 372 pages. Radia Perlman series in computer networking and security.
[4] Robert C. Seacord. 2013. *Secure Coding in C and C++* (2nd ed.). Addison-Wesley Professional.