Proposal for C2x

WG14 n2542

Title: Defer Mechanism for C

Author, affiliation:
Aaron Ballman, Self
Alex Gilding, Perforce
Jens Gustedt, Inria, France
Tom Scogland, Lawrence Livermore National Laboratory
Robert C. Seacord, NCC Group
Martin Uecker, University Medical Center Göttingen
Freek Wiedijk, Radboud Universiteit Nijmegen

Date: 2020-09-08

Proposal category: Feature

Target audience: Implementers

Abstract: Add a defer mechanism to C language to aid in resource management

Prior art: C, Go

Defer Mechanism for C

Reply-to: Robert C. Seacord (rcseacord@gmail.com)

Document No: n2542

Date: 2020-09-08

Defer Mechanism for C

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., parenthesis or HTML).

A separable feature of the defer mechanism is a panic/recover mechanism that allows error handling at a distance.

Table of Contents

Resource Management

A resource is a physical or virtual component with limited availability. Resources include memory, persistent storage, file handles, network connections, timers, and anything that exists in limited quantities. Resources also includes anything that itself contains or holds a limited resource. These resources need to be managed, meaning that they must be acquired and released. 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 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`**

Examples of C standard library functions that release resources include:

- allocated storage: **`free`**
- streams: **`fclose`**
- temporary file: **`fclose`**
- threads: **`thrd_join`**, **`thrd_detach`**
- thread specific storage: **`tss_delete`**
- condition variable: **`cnd_destroy`**

- condition variable: **`cnd_signal`**, **`cnd_broadcast`**
- mutexes: **`mtx_destroy`**, **`mtx_unlock`**

Annex K also contains many functions that parallel these functions and acquire/release resources. Matching acquire/release functions may also be defined by other standards such as POSIX or in user code. Functions that acquire and release resources may potentially be identified using attributes.

Acquiring Resources

Because acquiring a resource can fail when the supply of that resource has been exhausted, acquisition failures can and do happen. Consequently, safety-critical systems typically cannot depend on acquired resources that may not be available. In some systems, resources can be acquired during initialization and before any safety-critical operations commence (e.g., airplanes before takeoff). Security-critical systems may respond differently to resource acquisition failures, depending on their threat model. Some security-critical systems may treat any error as a potential attack and choose to abruptly terminate execution. Other systems that must remain available, may attempt to remain resilient to resource acquisition failures.

Acquiring resources can introduce potential failure points. Robust, reliable, and high-availability software systems require that these potential failures be managed in a consistent manner that minimally guarantees that the system terminates or continues execution in a known state. As a result, the dual problems of resource management and error handling are closely related, and both need to be considered in concert to produce usable solutions to both problems. Resource acquisition failures are recoverable, as resources may be released allowing execution to continue, but this approach often requires an extensive amount of forethought and effort and the system may continue execution in a degraded mode of operation [16]. Significant complexity can result in systems that attempt to recover from and continue execution following the failure of a resource acquisition.

Releasing Resources

Some resources are automatically released, while other resources must be manually released. For resources that are manually released, the programmer acquires the resource and explicitly releases the resource when it is no longer needed. Frequently, releasing a resource more than once without an intervening reacquisition will result in a defect. Resources, particularly storage, can be automatically released through various techniques. In C, each object has a storage duration that determines its lifetime. The following table shows when storage is released:

Storage Duration	**Released When**
Automatic	Block with definition is left (often only effective when function exits)
Thread	Thread exits

Static	Program exits
Allocated	Program exits, or memory is explicitly deallocated

Of these storage durations, allocated storage is of the greatest concern as this storage is typically acquired and released as required by the programmer. In these cases, the programmer cannot wait until the program exits because this resource may be exhausted. Other programming languages such as Java, C#, and Go use garbage collection to reclaim allocated storage that is no longer being used. Consequently, garbage collection is a specialized mechanism used to release one particular type of resource. Garbage collection eliminates some problems with manually releasing resources, but can have a suboptimal impact on performance. For example, garbage can accumulate in application and increase the memory usage high water mark. Collection can occur at unpredictable times, resulting in high CPU usage and the application appearing nonresponsive to the user. In particular, when garbage collection has five times as much memory as required, its runtime performance matches or slightly exceeds that of explicit memory management. However, garbage collection's performance degrades substantially when using a smaller heap. With three times as much memory, garbage collection runs 17% slower on average, and with twice as much memory, it runs 70% slower. Garbage collection is more susceptible to paging when physical memory is scarce. In such conditions, garbage collectors may suffer order-of-magnitude performance penalties relative to explicit memory management [6].

Other resources may corrupt the persistent execution environment such as a lock file, database, or file system. The C library also offers means to release resources at the end of a program (**`atexit`** and **`at_quick_exit`** handlers) and thread (**`tss_t`** destructors) execution.

For many resource types, the developer must explicitly release each resource when it is no longer required. In general, each acquired resource must be released once and only once. Resource release can sometimes fail. For example, it's possible for the C Standard **`fclose`** function to fail. When **`fclose`** writes the remaining buffered output, for example, it might return an error because the disk is full. Even if the user knows that the buffer is empty, errors can still occur when closing a file using the Network File System (NFS) protocol. On the other hand, the C Standard **`free`** function returns no value and has no way of indicating an error, but has undefined behavior if the argument does not match a pointer earlier returned by a memory management function, or if the space has already been deallocated by another call to **`free`** or **`realloc`**. All of these undefined behaviors can result in a corruption of heap structures that can be potentially exploited, and in general leave the execution in an undefined state.

The following tables show some resource types that can be managed by a C program, if they can fail on release, if this failure is reported, and if the resource is automatically released on thread exit or program exit.

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`**	✔	✔	✗	✗

For thread specific storage, we distinguish the acquire and release (that is, create and delete) of the key itself, which is per program execution, and the acquire and release of the individual thread specific data stored through that key, where a key-specific destructor is implicitly called when a thread exists.

The **`fclose`** and **`remove`** functions fail in a fundamentally different way from the **`free`** and **`tss_delete`** functions. The **`free`** and **`tss_delete`** functions have preconditions that can be satisfied by construction. A well constructed program could theoretically prevent these failures, although defects are common. The **`fclose`** and **`remove`** functions can fail as a result of resources not being available. Those are preconditions that can't be checked in a race-free manner through software alone.

Synchronization Primitives

The C language supports multiple threads of execution starting with C11. Threads themselves are resources that are created using **`thrd_create`** and disposed using **`thrd_join`** and **`thrd_detach`**. More important to this proposal, C11 also introduced condition variables and mutexes to act as synchronization primitives to help eliminate data races in concurrent code. There are numerous APIs that implement these or similar interfaces, including POSIX which specifies a set of interfaces for threaded programming commonly known as POSIX threads, or Pthreads.

The C Standard provides the **`mtx_init`** function which creates a mutex object. The **`mtx_init`** function returns **`thrd_success`** on success, or **`thrd_error`** if the request could not be honored. The **`mtx_destroy`** function releases any resources used by the mutex pointed to by mtx. The **`mtx_destroy`** function returns no value, and consequently cannot fail in a manner that can be programmatically detected. These functions behave as paired acquire / release functions that might be managed by a defer mechanism in a similar fashion to allocated storage or streams.

The C Standard also defines functions such as the **mtx_lock** function that blocks until it locks a specified mutex and the **mtx_unlock** function that unlocks a specified mutex. Both functions return **thrd_success** on success, or **thrd_error** if the request could not be honored. The following example illustrates how these functions are normally paired:

```
int do_work(void *dummy) {
  if (thrd_success != mtx_lock(&lock)) {
    return -1;
  }

  /* Critical section */

  if (thrd_success != mtx_unlock(&lock)) {
    return -2;
  }
  return 0;
}
```

It is critical for the proper functioning of a system that locked mutexes are properly unlocked. A common error is to return from the critical section without calling `mtx_unlock`. This error can result in a deadlock because the lock can no longer be acquired.

While these functions do not necessarily create nor destroy the resource, they do acquire and release the locks in a paired manner. Acquiring and releasing mutexes is a strong use case for a defer mechanism which can help ensure that locking and unlocking are properly paired.

Resources may be allocated for mutexes, depending on the backing implementation for `mtx_lock`. Objects of type `pthread_mutex_t` can be initialized with `PTHREAD_MUTEX_INITIALIZER`. On Linux, mutexes are often backed by a futex. When a futex is contended, there is an allocation for a hash table entry in the kernel[1] and other resources are being created in the kernel. For older versions of Windows, acquiring a `CRITICAL_SECTION` could result in the allocation of a Windows event object (similar to a semaphore).[2]

The C Standard provides the **cnd_init** function to create a condition variable and the **cnd_destroy** function releases all resources used by the specified condition variable. The **cnd_init** function returns **thrd_success** on success, or **thrd_nomem** if no memory could be allocated for the newly created condition, or **thrd_error** if the request could not be honored.The **cnd_destroy** function returns no value. These functions behave as paired acquire / release functions that might be managed by a defer mechanism in a similar fashion to mutexes.

[1] A futex overview and update
[2] https://devblogs.microsoft.com/oldnewthing/20140911-00/?p=44103

Security Concerns

Software security typically assumes an intelligent adversary that is working to compromise the security or possibly the availability of a system. 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 [1]. DoS attacks attempts frequently take the form of a resource-exhaustion attack that makes a computer resource unavailable or insufficiently available to the application. 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. MITRE maintains a list of common software and hardware weakness types called the *common weakness enumeration* (CWE) [2]. This general class of vulnerability is classified by MITRE as CWE-400: Uncontrolled Resource Consumption: “The software does not properly control the allocation and maintenance of a limited resource thereby enabling an actor to influence the amount of resources consumed, eventually leading to the exhaustion of available resources.” [3] In 2019, CWE-400 was ranked 20th of the 2019 CWE Top 25 Most Dangerous Software Errors [4]. This ranking represents the frequency of the issue as measured by the number of times a CWE is mapped to a CVE (common vulnerabilities and exposures) within NIST’s National Vulnerability Database (NVD). Another factor is a weakness severity, which is represented by the average *common vulnerability scoring system* (CVSS) score of all CVEs that map to a particular CVE.

A common error associated with manual memory management is deallocated memory more than once without an intervening allocation. This vulnerability class is described by *CWE-415: Double Free* [5] and 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. The NVD contains 210 instances of CWE-415 reported in the period from August 2011 to June 2020. This is likely grossly underreported since before 2016, because NVD only used ~19 different CWEs.

Error Handling

The separable panic/recover mechanism is used to perform error handling at a distance and is consequently similar to exception handling in C++ (see Appendix B for more information).

Exception-safety is a concept developed primarily by David Abrahams [11] where a component exhibits *reasonable* behavior when an exception is thrown during its execution. Exception-safety evolved from the development of STLport, a multiplatform ANSI C++ Standard Library implementation[3]. Consequently, the concept originally evolved around C++ exceptions but can be generalized to a variety of languages and error handling mechanisms.

Reasonable behavior for error handling means that resources are not leaked, and that the program remains in a well-defined state so that execution can continue. In most cases, it also includes the expectation that when an error is encountered, it is reported to the caller.

A software component might provide one of the following safety guarantees:

1. The basic guarantee: that the invariants of the component are preserved, and no resources are leaked.
2. The strong guarantee: that the operation has either completed successfully or indicated an error, leaving the program state exactly as it was before the operation started.
3. Failure transparency: operations are guaranteed to succeed and satisfy all requirements even in exceptional situations. If an error occurs, it will be handled internally and not observed by clients.

The basic guarantee is a simple minimum standard for error handling. It says simply that after an error, the component can still be used as before. Importantly, the preservation of invariants allows the component to be destroyed, potentially as part of stack-unwinding. This guarantee is actually less useful than it might at first appear. If a component has many valid states, after an exception we have no idea what state the component is in; only that the state is valid. The options for recovery in this case are limited: either destruction or resetting the component to some known state before further use.

The *strong* guarantee provides full *commit-or-rollback* semantics. In the case of C++ standard containers, this means, for example, that if an exception is thrown all iterators remain valid. We also know that the container has exactly the same elements as before the exception was thrown. A transaction that has no effects if it fails has obvious benefits: the program state is simple and predictable in case of an exception. Providing the strong guarantee can often have substantial performance implications. In C++98, the dynamic array class, **`std::vector`**, was specified to have the strong exception guarantee when performing **`push_back`**. At the time, the strong guarantee could be provided *for free*. *Move semantics* were added to C++11, and the strong guarantee came at a cost for important **`push_back`** use cases.

Failure transparency is the strongest guarantee of all, and it says that an operation is guaranteed not to fail: it always completes successfully. In C++, this guarantee is necessary for most destructors, and indeed the destructors of C++ standard library components are all guaranteed not to throw exceptions. No-fail guarantees are also important for move and swap operations, which are often used to commit a transaction. To provide failure transparency, you often need some number of no-fail building blocks. Move and swap operations are often used to commit a transaction. To provide failure transparency, no-fail building blocks are required.

[3] http://www.stlport.org/doc/exception_safety.html

One goal of this proposal is to allow developers to provide all three levels of safety guarantees using the defer mechanism.

Non-critical Failure Performance

There are a great many applications where the performance on the failure path is of low importance.

In many interactive applications, a user interaction will often cause some operation to take place, and if that operation fails, then a dialog, or some other kind of user interaction is shown (e.g., opening a document of some kind). The value of the program is the same whether the dialog shows up in 1 millisecond or 50 milliseconds.

There are non-interactive applications that care a great deal about latency on the success path, but don’t care about performance on the failure path. In the failure cases, no transaction needs to happen, so it is acceptable to take a longer amount of time, so long as the latency of the next success path operation is unaffected.

There are non-interactive simulation and modeling applications (e.g. high-performance computing) where no errors occur during a successful simulation, and errors only occur during an unsuccessful run. When these errors happen, it normally indicates some kind of failure that requires human intervention, like adjusting the starting parameters, or freeing up hard disk space, or plugging a network cable back in. A few extra milliseconds is perfectly acceptable in these circumstances.

Some non-interactive HPC applications have specific code to handle run-time failures, namely they produce snapshots of the whole application, such that it may then be restarted later. Such snapshots usually take at least several seconds, and so spending some milliseconds for launching such a mechanism is neglectable.

Conversely, there are safety-critical systems where the failure path is the important one to be optimized for. For instance, when recovering from a failure that sends the elevator hurtling towards the ground, the success path's performance isn't nearly as concerning as the failure path which arrests the fall.

The N+1 Problem

There is no shortage of error handling mechanisms in the standard library and in the wild today. The "N" is already a large number. C uses integer error codes and `errno`.

Even if the C standard does not add new error handling mechanisms, the community will likely do so.

It is commonplace for users to need to translate the results of various error handling mechanisms into the error handling mechanism that their program uses. The translation is not difficult, though it is verbose and can be error prone.

Creating a new error handling mechanism does not make the old mechanisms go away. If you had N mechanisms before, a new mechanism, no matter how superior a choice, will leave you with N+1 mechanisms.

A new error handling mechanism could change how we teach error handling, and how we write error handling in new code. This has happened in the past, and could happen again. There is very little educational effort spent on **`errno`** best practices, and little new C code is written with **`errno`** as the primary error handling mechanism. The main interaction with **`errno`** is in handling the errors and translating them to newer mechanisms. A new error handling mechanism can aspire to replace other mechanisms as well as **`errno`** has been replaced.

Do we want a defer statement?

A **`defer`** statement defers the execution of a *deferred statement* until the containing guarded block terminates. A defer statement is associated with its nearest enclosing guarded block or function body. Its deferred statement is sequenced in last-in-first out (LIFO) order after all statements that are contained in that guarded block and before the guarded block itself terminates. The primary use of a defer statement is to release acquired resources independent of how a block exits. Consequently, deferred statements should avoid allocating new resources and must provide failure transparency.

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

statement:

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

defer-statement:

defer *statement*

In the presence of deferred statements, the C library functions **`exit`** and **`thrd_exit`** gain additional behavior. Before the normal processing of the termination event, they trigger an execution of all deferred statements for the current thread by unwinding the stack (see Appendix I). This processing of the deferred statements cannot be stopped by calling **`recover`**.

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.[4]
- 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.

Deferred statements can include a **`continue`** statement that is associated with an iteration statement that is a substatement of the deferred statement.
There has been some discussion about disallowing **`guard`** and **`defer`** statements within defer statements to limit complexity. However, these are simply control structures and because they may be used within library functions called from deferred statements, violations would be difficult to diagnose.

The deferred statement is executed as often as the **`defer`** statement is encountered, but there are open questions on some aspects of the proposal such as whether deferred statements require storing state at runtime or not, whether there should be improved error handling capabilities added or not, whether to use implicit or explicit markings to denote when deferred statements are executed, etc.

Should defer statements be static or dynamic?

A guarded block can contain multiple **`defer`** statements. While deferred statements are generally intended to be sequenced in the reverse order in which the defer statements are encountered during execution, there are some open design questions that can be broadly categorized as the static or dynamic approach.

The goal of the dynamic approach is to match programmer expectations based on control flow. In the following code, for example:

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

[4] Calling **`abort`** may be preferred in high security situations where the risk of triggering remote code execution must be reduced. Any negative interactions with thread local destructors should be avoided.

The dynamic approach suggests that **`whatever(x)`** is deferred only if **`x`** evaluates to a non-zero value at runtime. For iteration statements, the deferred statement is pushed for each iteration of the loop in which the **`defer`** statements are encountered. For this example, **`n`** separate deferred calls to **`printf`** are pushed on the deferred execution stack. Note that the declaration of **`i`** has to be on the same level as the **`guard`** block, because otherwise it would be out of scope when these deferred statements are executed, and that we have to decrement that variable within the deferred statement to see it decreasing.

As the name implies, the dynamic approach suggests that resources need to be allocated at runtime which implies additional runtime overhead and creates the possibility that these operations might fail.

Loops may also be constructed with **`goto`** statements and labels, as long as they do not break out of a deferred or guarded block.

The goal of the static approach is to statically allocate the required resources at compile time, eliminate the possibility that deferred statements may fail at runtime, and to be optimally efficient.

For the **`if`**, the compiler provides a slot in the same way that it would for an unconditional defer. Loops are similar. The static approach can provide one static slot for the defer statement that appears inside the loop.

A **`defer`** statement could merely record whether it has been triggered or not. One bit of information per defer statement in the activation record. If it has been triggered, the code is executed once at the end of the corresponding guarded block. If the loop is not entered, the deferred statement is not executed.

It is possible that the **`defer`** statement may be evaluated in something other than the lexical order because of a **`goto`** statement, for example. We could simply specify that **`defer`** statements are executed in reverse lexical order.

Should object values be captured?

An open question if the values of objects accessed in deferred statements should be captured when the deferred statement is encountered or their latest values read when the deferred statements are evaluated?

Any identifier that is accessed in the deferred statement has to be visible for the **`defer`** statement and its scope must extend to at least the end of the nearest enclosing guarded block or function body if there is no guarded block. One perspective is that these objects will have their current values at the point the deferred statements are executed. Identifiers of object type that are visible by the **`defer`** statement and for which the underlying objects are alive at the end of the execution of the guarded block remain accessible, and the objects have their most recently written values.

A prominent example why access to local variables with their value at deferred execution may be desired is reallocating the storage by using **realloc**:

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

However, if the programmer assumes that the values are captured, it is easy to write incorrect code such as the following:

```
{
  struct s *ptr = malloc(sizeof(struct s) * 10);
  defer free(ptr);

  for (int i = 0; i < 10; ++i) {
    whatever(ptr++);
  }
}
```

The use of **errno** in the following code snippet is also problematic:

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

The value of **errno** must be examined before a subsequent call that may set the global **errno** is invoked.

We created a simple Twitter poll to test programmer expectations:

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;
}
```

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%). One interesting thing to note is that of respondents who gave a rationale for their vote, respondents known to have a strong C++ background seemed to gravitate towards using the value when the defer statement is encountered, which suggests there may be some interesting implicit bias.

A general solution to this problem is the introduction of lambdas to the C language [13] with both copy and reference semantics. A more specific solution is to use a second defer statement syntax which supplies an identifier list of variables to capture by value.

Go's main use case also captures object values in a similar manner to lambdas and as described in Appendix E.

Do we want the `guard` keyword?

Deferred statements appear in guarded blocks, but it is an open design question if these guarded blocks need to be explicitly indicated by a `guard` keyword. If we don't want a `guard` keyword there is a language design choice to make where to attach the execution of deferred statements. One option is to execute the deferred statements at the end of the closest enclosing scope of the defer statement. Another option would be to execute the deferred statements only at the end of the function body (as Go does). Using explicit `guard` blocks or the second option allows for dynamic collection of deferred statements during the execution of a function (see above), while the first option imposes a strictly static execution at the end of each scope.

In the keyword approach, the *compound statement* of a `guard` statement is called a *guarded block*. A `guard` statement indicates that any deferred statements within the guarded block will be executed just before the guarded block terminates.

guard-statement:

  guard *compound-statement*

Guarded statements may contain zero or more defer statements. Deferred statements are executed just before the guarded block terminates. Blocks may terminate as the result of normal control flow execution or because a call to `exit`, `_Exit`, `quick_exit`, `thrd_exit` or `panic` is issued, or because an abnormal runtime condition is met that triggers a panic.

Guarded blocks shall not call functions that call **`longjmp`** to jump to a location outside the guarded block and the state of any **`jmp_buf`** that has been last set by a call to the **`setjmp`** macro within the guarded block is indeterminate, once the guarded block has been left.

An alternative to the **`guard`** statement is to execute deferred statements at the termination of the current scope. For **`if`** and **`switch`** statements, the selection statement is a block whose scope is a strict subset of the scope of its enclosing block. Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement. For **`for`**, **`do`**, and **`while`** loops, the iteration statement is a block whose scope is a strict subset of the scope of its enclosing block. The loop body is also a block whose scope is a strict subset of the scope of the iteration statement. Using scope to determine when to execute deferred statements is a similar design to how Resource Acquisition Is Initialization (RAII) works in C++. Consequently, C++ programmers seem more likely to assume deferred statements execute at the end of the current scope.

The **`guard`** statement allows for a library implementation. Foregoing the possibility of a library implementation, a possible design choice could be to eliminate the **`guard`** statement as it would eliminate the need for an additional reserved keyword and the requirement for programmers to create guarded blocks around deferred statements. If the **`guard`** statement is not used, the proposed changes to the behavior of the **`break`** statement would likely be eliminated as well (see Appendix K).

Following is a simple example of using the **`defer`** statement without the **`guard`** keyword and an attachment of the deferred statement to the end of the surrounding compound statement:

```
void *ptr = 0;
if (ptr = malloc(12)) {
  defer free(ptr);
  // Use ptr
} // free ptr here
```

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
```

Without the **guard** keyword, and in a model where all scopes would be guarded, this would execute the **defer** directly at the end of the scope of the **if** statement, and the use of **ptr** after that would be undefined. 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
} // free ptr here
```

The **guard** keyword allows the programmer to write code with lifetime issues that would be impossible with the scope model. Consider this contrived example where the defer statement uses a local variable **i** which is outside of its lifetime when the deferred statement executes:

```
guard {
  if (something) {
    int i = 0;
    ...
    defer printf("%d", i);
  }
} // uses i outside of its lifetime
```

But such a usage of variables that will be out-of-scope when the deferred statement is executed can be made a constraint violation, and so we expect that compilers will be able to diagnose such situations.

The **guard** keyword is not particularly useful for the following code:

```
if (whatever) guard {
  defer whatever_else();
}
```

However, the **guard** keyword may be required to express the following code because a compound statement as a loop body is its own scope, so using the scope of the **for** loop body would result in executing the deferred statement on each iteration of the loop:

```
guard {
  int i;
  for (i = 0; i < 10; ++i) {
    defer whatever(i);
  }
} // all deferred statements get run here
```

Making the guarded block explicit is strictly more expressive and allows the programmer more control over when the deferred statements are evaluated. However, the explicit syntax adds an additional keyword. Eliminating the `guard` keyword provides a more terse syntax and eliminates any issues with introducing a new keyword, but at the cost of a grammatical ambiguity where deferred statements would be attached.

Do we want a panic/recover mechanism?

The primary purpose of a defer mechanism 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.

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 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 will *unwind* the caller's stack (see Appendix I) and execute all deferred statements registered in that stack frame. It will execute until a `recover` expression is encountered or all deferred statements have executed.

In the latter case, the handler `handler` is called as if by `handler(code)` and is typically used to terminate the current thread or the whole program execution. If a call to `recover` is evaluated and the value of `code` is equal to 0, the stack will continue to unwind following execution of the remaining deferred statements of the current defer statement. If the value of code is not equal to 0, unwinding will cease and control will continue and the statement following the block or scope containing the deferred statement will be evaluated.

The argument **`handler`**, if provided, shall be a pointer to a function that terminates the current thread or the program. If the argument **`handler`** is omitted an implementation-defined function is called that is expected to terminate execution. If it is a null pointer, if it exists, a handler that had been previously established by another call to panic on that thread and that has been stopped by a call to **`recover`** is used. If no such handler has been established the same handler is used as if the argument were omitted. It is undefined behavior if the panic handler resumes execution instead of terminating the thread or program.

The first argument is an error code of type **`int`**. The error code is recommended to be:

- Negative to indicate a system defined error condition as if for **`errno`**, including an error condition triggered by the implementation.
- Zero to indicate a normal termination of the containing guarded block, calling thread or the whole program execution.
- Positive to indicate an application supplied error number.

Following an invocation of the **`panic`** macro until the panic handler completes execution or a successful recovery action, the program is said to be *panicking*.

The **`panic`** macro may output implementation-defined information about the cause of the panic and a trace of the respective contexts traversed during unwinding to **`stderr`**. The implementation specific use of output functions notwithstanding, the use of the **`panic`** macro is asynchronous signal safe and may be called from a signal handler. Whether or not such a use from a signal handler triggers the execution of deferred statements is implementation-defined.

The second form of the **`panic`** macro:

```
_Noreturn void panic(int code, panic_handler_t handler);
```

can be used to specify a pointer to a function that terminates the current thread or the program. C programmers currently have the ability to terminate a thread by calling the C Standard **`thrd_exit`** function and a program by calling the C Standard **`exit`**, **`quick_exit`**, **`_Exit`**, or **`abort`** functions. The signature specified by the **`panic_handler_t`** type supports all these options with the exception of the **`abort`** function, which could be called from a wrapper.

Panicking is meant to be a *gentle* way to exit a program, allowing resources to be released. Consequently, using an abrupt mechanism such as the **`abort`** function is generally not recommended. A recommended practice is to use the **`exit`** function.

Both forms of the **`panic`** macro accept an **`int`** argument because that is compatible with the C Standard exit functions that receive an **`int`** and error specifications specified as **`errno`**, which is also an **`int`**.

A primary motivation of allowing a user to specify a panic handler is to let the user distinguish between termination of the thread or the program.

For example, a systematic use

`defer mtx_unlock(&bla);`

(or the equivalent for whatever thread implementation is used)

would gain safety if paired with a use of

`panic(-EVERYBAD, thrd_exit)`

to indicate thread local error conditions. This would keep the process alive and would guarantee to allow all mutexes to unlock when terminating the thread.

A similar effect could also be achieved by using **`thrd_exit(-EVERYBAD)`** alone, as we now also impose that it executes deferred statements. But the **`panic`** mechanism is a bit more debugging friendly as it allows to print call traces along and things like that.

One use is that a comparison to that function pointer can distinguish different debugging behavior of the unwind mechanism. This can be achieved by querying the panic handler of the current thread.

The **`recover`** function has the following signature:

```
    #include <stddefer.h>
    int recover(void);
```

A call to the **`recover`** function shall only appear within a deferred statement.

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. Neither the **`_Exit`** or **`abort`** functions would cause an unwind.

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. The **code** argument can be assigned the recovered value to preserve the previous error code or set to a new value. In the latter case, the previous (possibly original) reason for the failure is lost. If a terminating function is supplied as the **func** argument to the **panic** macro, the function is installed as the new panic handler. If no **func** argument is supplied, or a null pointer is supplied as the **func** argument, the previously established panic handler is retained. It is currently undefined what happens if the **panic** macro is invoked in a **defer** statement before the **recover** function is invoked, or after the recover function recovers a zero error condition.

The fragment

```
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);
  }
}

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.");
```

```
   }
}
```

Contains a function **`f`** containing a **`defer`** statement which contains a call to the **`recover`** function. Function **`f`** invokes function **`g`** which recursively descends before panicking when the value of **`i > 3`**. Execution of **`f`** produces the following 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.
```

Summary

The defer mechanism in C provides a general mechanism for deferring the execution of paired operations on a first in last out basis. This mechanism can be used to improve resource management, the acquisition and release of synchronization primitives, and other applications that would benefit from a user-controlled execution stack.

This basic mechanism can be enhanced with a panic/recover mechanism that allows errors to be handled at a distance from the original failure.

Appendix A: Resource Management and Error Handling in C

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 the duplication and additional complexity.

Programs based on error codes are usually littered with statements of the form:

```
if (FAILED(err)) { return err; }
```

Programs based on error codes where the error code is the return value also use a boilerplate where they need to declare out-parameters in advance of a function call, and even more boilerplate when results of functions need to be composed.

Error codes can be accidentally ignored. Doing "the right thing" requires the programmer to write additional code to avoid "the wrong thing". The `[[nodiscard]]` attribute substantially improves this situation.

One solution is to use nested if statements, which can also become difficult to read if nested too deeply. Alternatively, a `goto` chain can be used to release resources.

```
int do_something(void) {
  FILE *file1, *file2;
  object_t *obj;
  int ret_val = 0; // Initially assume a success
  file1 = fopen("a_file", "w");
  if (file1 == NULL) {
    ret_val = -1;
    goto FAIL_FILE1;
  }
  file2 = fopen("another_file", "w");
  if (file2 == NULL) {
    ret_val = -1;
    goto FAIL_FILE2;
  }
  obj = malloc(sizeof(object_t));
  if (obj == NULL) {
    ret_val = -1;
    goto FAIL_OBJ;
  }
  // Operate on allocated resources

  // Clean up everything
  free(obj);
FAIL_OBJ: // Otherwise, close only streams we opened
  fclose(file2);
FAIL_FILE2:
  fclose(file1);
FAIL_FILE1:
  return ret_val;
}
```

There are well-known and well-studied examples of failures using a goto chain, such as the following example that resulted in the Heartbleed OpenSSL security vulnerability:

```
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
```

The programmer in this example has accidentally repeated the line:

```
goto fail;
```

The first `goto fail` statement executes if the if `err` is non-zero. This causes the code to fail with error, and the entire TLS connection fails.

The second, unintended `goto fail` is always executed, in this case, when `err` is zero and there is no error to report. The result is that the code jumps over the call to the `sslRawVerify` function, and exits the function. This causes an immediate “exit and report success”, and the TLS connection succeeds, even though the verification process hasn’t actually taken place.

GCC and Clang Dialects

The GNU C dialect (as implemented by GCC and Clang) provides an attribute which can be used to enable automatic destruction on end of scope for an annotated block-scope object:

```
#include <stdio.h>

static void fp_close (FILE ** fpp) { fclose (*fpp); }

extern void do_work (FILE *);

void foo (char const * name) {
      __attribute__((cleanup(fp_close)))
      FILE * fp = fopen (name, "r"); // run fp_close after do_work

      do_work (fp);
}
```

This takes advantage of existing infrastructure used to compile C++ code that uses RAII (Resource Acquisition Is Initialization). "Cleanup" functions are associated with an automatic variable and are guaranteed to run on exit from the containing scope, either as part of normal execution, or under stack-unwinding conditions (when a C++-like exception has been thrown). This is primarily useful for making C code that is expected to interoperate with C++ more robust, but it does coincidentally mean resource release code is placed closer to the point of declaration/allocation in the source.

This provides real-world implementation experience of unwinding support for C in industrial-quality compilers. Because the code generation shares optimization with the C++ implementation's support for RAII, which is well-understood and already highly optimized, the cleanup calls are able to be inlined, and the code is efficient on the non-exceptional path: the cleanup function will usually be completely inlined, as though the C code had been written with a "traditional" explicit cleanup call on end-of-scope. Additional code is generated out-of-line for the unwinding path, so that the common-case can take advantage of the feature at zero-cost.

This can be combined with additional local-function extensions to create a construct with deferred error-handling blocks, superficially resembling Go's **`defer`** (GNU C does not provide any catch/recover facility):

```
#if defined __clang__  // requires -fblocks (lambdas)
```

```
static void cleanup_deferred (void (^*d) (void)) {
 (*d)();
}
#define defer(...)                              \
  __attribute__((__cleanup__ (cleanup_deferred)))\
  void (^DF_##__LINE__) (void) = ^__VA_ARGS__

#elif defined __GNUC__ // nested-function-in-stmt-expression

static void cleanup_deferred (void (**d) (void)) {
 (*d)();
}
#define defer(...)                                      \
  __attribute__((__cleanup__ (cleanup_deferred)))    \
  void (*DF_##__LINE__) (void) = ({                     \
      void DF_##__LINE__##_impl (void) __VA_ARGS__      \
      DF_##__LINE__##_impl; })

#endif

extern int bar (int);

void foo (void) {
      bar(1);
      defer ({ bar(3); });
      bar(2);
}
```

In this case, the cleanup function just runs the local function, with its captured action representing the deferred statement. Both code generators inline the deferred statement at the end of the block, just as efficiently as if it had been written there explicitly. If the code is compiled with the exception-awareness flag, separate code is again generated to place the deferred action on the unwinding path without affecting the efficiency of the common case.

Appendix B: Resource Management and Error Handling in C++

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers.

To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a **`try`** block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

An exception is thrown by using the **`throw`** keyword from inside the try block. Exception handlers are declared with the keyword **`catch`**, which must be placed immediately after the **`try`** block:

```
// exceptions
#include <iostream>
using namespace std;
 int main(void) {
  try {
    throw 20;
  }
  catch (int e) {
    cout << "An exception occurred. Exception Nr. "
         << e << endl;
  }
  return 0;
}
```

In C++, the *resource acquisition is initialization* (RAII) idiom is used extensively. Resources are controlled by an object that links the resource's lifetime to the objects. Using RAII, every resource allocation should occur in its own statement (to avoid sub-expression evaluation order issues). The object's constructor immediately puts the resource in the charge of a resource handle. The object's destructor releases the resource. Copying and heap allocation of the resource handle object are carefully controlled or outright denied.

If the "use f" part of **`old_fct`** throws an exception or returns the file isn't closed.

```
  void old_fct(const char* s) {
    FILE * f = fopen(s, "r");    // open the file "s"
    // use f
    fclose(f);  // close the file
  }
```

If the "use f" part of **`fct`** throws an exception, the destructor is still called and the file is properly closed.

```
  void fct(string s) {
    // File_handle's ctor opens file "s"
    File_handle f(s, "r");
    // use f
  } // File_handle destructor closes the file here
```

RAII can be used to write C++ code that ensures resources are properly released:

```
class IntArray {
public:
  int *ptr;
  IntArray() : ptr(new int[100]) {}
  ~IntArray() { delete [] ptr; }
};
{
  IntArray x;
  // work with x
  IntArray y;
  // work with x and y
} // x, y cleaned up, even if exception.
```

If the "use f" part of **`old_fct`** throws an exception or returns the file isn't closed.

C++ is the closest analog to C. C++ **`catch`** clauses have as arguments a parameter and its type. The parameter is set by the system for standard exceptions and by the programmer for programmer-defined exceptions generated with the **`throw`** construct; the **`catch`** clause is matched to the **`throw`** by the type of the thrown object. There is also a **`catch(...)`** syntax to catch all unspecified exceptions.

WG14 N1841 - Alternate Exception Handling Syntax for C [7] contemplates the use of C++ exception handling mechanisms to express IEEE 754-2008 alternate exception handling in C.

P1095R0/N2289: Zero overhead deterministic failure A unified mechanism for C and C++ [15] proposes a universal mechanism for enabling C speaking programming languages to tell C code, and potentially one another, about failure and disappointment. This paper was discussed at the Pittsburgh meeting with the following results:

- Does this group like the direction of N2289: Yes 15/2/2.

- Do we want to see this proposal in two parts (a function returns an extra bit for failure, and the second part being types to return extra information)? 6/7/6. Noconsensus.

SO/IEC JTC1 SC22/WG14 AND INCITS J11 APRIL 2008 MEETING MINUTES
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1343.pdf

10.11 N1298 - Cleanup and try finally for 'C' (Stoughton) Both these constructs have value, and recommendation is to proceed with both. PJ believes we should also add the constructor and destructor attributes to this pair. Nick does not have the bandwidth to proceed here, and would like to hand this off to someone else! Nick: The constructor/destructor attributes require linker/loader and other runtime support and do not belong in a language standard. General support for try-finally. Should there be exception handling too? try-except-finally? Nick said this was never something he planned. Structured exception handling is widely used and understood, and a part of the MSVC compiler. It is also imperfect, and has known flaws. Try-finally is a well understood metaphor. Exception adds problems. C++ compatibility make it worse. Adding exception handling needs words about how an exception is thrown. If exceptions were limited to purely synchronous throws, it might be palatable. Bill Seymour agrees we should talk about exceptions, but it is a separate topic. Tom answered Robert's desire to see exceptions in C ... from certain points of view they close security vulnerabilities (people not testing return values). Nick points out that adding exceptions will break the existing library, and will need a whole new parallel library that is exception based. Still will be problems with old code. PJ feels try-finally is a better solution than cleanup, and Nick agrees he's never been in the position where he has had a choice.

Straw polls:

Who wants to see a fully-formed proposal for try-finally:

For: 11

Against: 0

Abstain: 7

Similar for cleanup attribute:

For: 2

Against: 9

Abstain: 7

Add some kind of exception handling for C:

For: 2

Against: 9

Abstain:

Boost.ScopeGuard

The **`Boost.ScopeGuard`** library fully emulates D's scope guard statement feature via C++17's `std::uncaught_exceptions()` with no extra dynamic memory allocations and terse syntax via class template deduction.[5]

Given a resource that lacks built-in RAII capability, e.g. a C-style `FILE*`, `boost::scope_guard` lets you manage its lifetime in an exception-safe and localized manner:

```
FILE* f = std::fopen("my/file", "r");
if (!f) { throw std::runtime_error("failed to open file"); }
boost::scope_guard my_guard = [&]{
    // Invoked when `my_guard` leaves scope, exactly when `f`'s
    // destructor would have been called if it had one.
    std::fclose(f);
};
// ...some code that can potentially throw exceptions...
```

A scope guard can also accept any number of additional arguments that will be passed to its function upon invocation. This means you can often use cleanup functions directly, without wrapping them in lambdas:

```
boost::scope_guard my_guard{std::fclose, f};
```

Function and arguments are stored by value by default, you can store them by reference via `std::ref` and `std::cref`:

```
std::thread my_thread(compute_something);
boost::scope_guard my_guard{&std::thread::join, std::ref(my_thread)};
```

[5] https://github.com/yuri-kilochek/boost.scope_guard

Having to name all your scope guard objects so that they don't conflict with each other quickly gets tiresome, so there is a `BOOST_SCOPE_GUARD` macro that does this automatically (however be aware of its limitations):

```
BOOST_SCOPE_GUARD {std::fclose, f};
```

Naturally, lambdas are supported as well:

```
BOOST_SCOPE_GUARD [&]{ my_thread.join(); };
```

Regular `boost::scope_guard` always invokes its stored cleanup function upon destruction, which may not be desirable. There is also `boost::scope_guard_failure` that invokes its stored cleanup function *only* when it is being destroyed due to stack unwinding (i.e. when an exception is thrown) and `boost::scope_guard_success` that invokes its stored cleanup function *only* when it is being destroyed due to flow of control leaving the scope normally. Naturally, there are corresponding `BOOST_SCOPE_GUARD_FAILURE` and `BOOST_SCOPE_GUARD_SUCCESS`.

Appendix C: Resource Management and Error Handling in D

Expressions that must always be executed are written in the **`finally`** block, and expressions that must be executed when there are error conditions are written in **`catch`** blocks.[6]

Some of the variables that these blocks need may not be accessible within these blocks:

```
void foo(ref int r) {
    try {
        int addend = 42;

        r += addend;
        mayThrow();

    } catch (Exception exc) {
```

[6] http://ddili.org/ders/d.en/scope.html

```
        r -= addend;            // ← compilation ERROR
    }
}
```

That function first modifies the reference parameter and then reverts this modification when an exception is thrown. In normal block-scoping, `addend` is accessible only in the `try` block, where it is defined.

The `scope` statements have similar functionality to the `catch` and `finally` scopes but they are better in many respects. Like `finally`, the three different `scope` statements are about executing expressions when leaving scopes:

Scope guards allow executing statements at certain conditions if the current block is left:

- `scope(exit)` will always call the statements
- `scope(success)` statements are called when no exceptions have been thrown
- `scope(failure)` denotes statements that will be called when an exception has been thrown before the block's end

Using scope guards makes code cleaner and allows resource allocation and clean up code to be placed next to each other.[7] These little helpers also improve safety because they make sure certain cleanup code is *always* called independent of which paths are actually taken at runtime.

The D `scope` feature effectively replaces the RAII idiom used in C++ which often leads to special scope guard objects for special resources.

Scope guards are called in the reverse order they are defined.

Scope Guard Statement

ScopeGuardStatement:

scope(exit) *NonEmptyOrScopeBlockStatement*

scope(success) *NonEmptyOrScopeBlockStatement*

scope(failure) *NonEmptyOrScopeBlockStatement*

The *ScopeGuardStatement* executes *NonEmptyOrScopeBlockStatement* at the close of the current scope, rather than at the point where the *ScopeGuardStatement* appears. [8] `scope(exit)` executes *NonEmptyOrScopeBlockStatement* when the scope exits normally or when it exits due to exception unwinding. `scope(failure)` executes

[7] https://tour.dlang.org/tour/en/gems/scope-guards
[8] https://dlang.org/spec/statement.html#scope-guard-statement

NonEmptyOrScopeBlockStatement when the scope exits due to exception unwinding. `scope(success)` executes *NonEmptyOrScopeBlockStatement* when the scope exits normally.

If there are multiple *ScopeGuardStatement*s in a scope, they will be executed in the reverse lexical order in which they appear. If any scope instances are to be destroyed upon the close of the scope, their destructions will be interleaved with the *ScopeGuardStatement*s in the reverse lexical order in which they appear.

```
write("1");
{
    write("2");
    scope(exit) write("3");
    scope(exit) write("4");
    write("5");
}
writeln();
```

writes:

12543

```
{
    scope(exit) write("1");
    scope(success) write("2");
    scope(exit) write("3");
    scope(success) write("4");
}
writeln();
```

writes:

4321

```
struct foo {
    this(string s) { write(s); }
    ~this() { write("1"); }
}

try {
    scope(exit) write("2");
    scope(success) write("3");
    Foo f = Foo("0");
    scope(failure) write("4");
    throw new Exception("msg");
    scope(exit) write("5");
    scope(success) write("6");
    scope(failure) write("7");
}
catch (Exception e) { }
writeln();
```

writes:

0412

A **`scope(exit)`** or **`scope(success)`** statement may not exit with a **`throw`**, **`goto`**, **`break`**, **`continue`**, or **`return`**; nor may it be entered with a **`goto`**.

Appendix D: Resource Management and Error Handling in Java

Java has an exception handling mechanism similar to C++, with the primary difference being that Java provides checked exceptions that are enforced to some degree by the compiler. In Java, non-memory resources are reclaimed via explicit **`try/finally`** blocks instead of the RAII idiom. The **`finally`** block always executes when the **`try`** block exits. This ensures that the **`finally`** block is executed even if an unexpected exception occurs. But **`finally`** is useful for more than just exception handling—it allows the programmer to avoid having cleanup code accidentally bypassed by a **`return`**, **`continue`**, or **`break`**. Putting cleanup code in a **`finally`** block is always a good practice, even when no exceptions are anticipated.

The following example uses a Java **`try-catch-finally`** block in an attempt to close two resources.

```
public void processFile(String in, String out)
  throws IOException{
  BufferedReader br = null;
  BufferedWriter bw = null;
  try {
    br = new BufferedReader(new FileReader(in));
    bw = new BufferedWriter(new FileWriter(out));
    // Process the input and produce the output
  }
  finally {
    try {
      if (br != null) {
        br.close();
      }
      if (bw != null) {
        bw.close();
      }
    }
    catch (IOException x) {
      // Handle error
    }
  }
}
```

This code has a defect, however, in that an exception can be thrown when **`br`** is closed, resulting in **`bw`** not being closed. This defect can be repaired by introducing a second finally block to guarantee that **`bw`** is properly closed even when an exception is thrown while closing **`br`**. However, Java 7 introduced the **`try-with-resources`** statement (see the JLS, §14.20.3, "try-with-resources" [10]), which simplifies correct use of resources that implement the **`java.lang.AutoCloseable`** interface, including those that implement the **`java.io.Closeable`** interface.

Using the **`try-with-resources`** statement avoids problems that can arise when closing resources with an ordinary **`try-catch-finally`** block, such as failing to close a resource because an exception is thrown as a result of closing another resource, or masking an important exception when a resource is closed.

This following on uses a **`try-with-resources`** statement to appropriately release the resources associated with both **`br`** and **`bw`**:

```
public void processFile(String in, String out) throws IOException{
  try (BufferedReader br = new BufferedReader(new FileReader(in));
      BufferedWriter bw = new BufferedWriter(new FileWriter(out))) {
    // Process the input and produce the output
  } catch (IOException ex) {
    System.err.println("thrown exception: " + ex.toString());
    Throwable[] suppressed = ex.getSuppressed();
    for (int i = 0; i < suppressed.length; i++) {
      System.err.println("suppressed: " + suppressed[i].toString());
    }
    }
}
```

The **`try-with-resources`** statement is not directly implementable in C, because resources are not as uniform as Java resources that implement the **`java.lang.AutoCloseable`** interface.

Java is a garbage-collected language. Memory is a special class of resource on these systems that is cleaned up or deallocated using a special mechanism.

Appendix E: Resource Management and Error Handling in Go

A defer statement in Go invokes a function whose execution is deferred to the moment the surrounding function returns, either because the surrounding function executed a `return` statement, reached the end of its function body, or because the corresponding goroutine is panicking.

```
DeferStmt = "defer" Expression .
```

The expression must be a function or method call; it cannot be parenthesized. Calls of built-in functions are restricted as for expression statements.

Each time a **`defer`** statement executes, the function value and parameters to the call are evaluated as usual and saved again but the actual function is not invoked. Instead, deferred functions are invoked immediately before the surrounding function returns, in the reverse order they were deferred. That is, if the surrounding function returns through an explicit **`return`** statement, deferred functions are executed after any result parameters are set by that return statement but before the function returns to its caller. If a deferred function value evaluates to **`nil`**, execution panics when the function is invoked, not when the **`defer`** statement is executed.

For instance, if the deferred function is a function literal and the surrounding function has named result parameters that are in scope within the literal, the deferred function may access and modify the result parameters before they are returned. If the deferred function has any return values, they are discarded when the function completes.

```
lock(l)
defer unlock(l)  // unlock happens before enclosing function returns
```

```
// prints 3 2 1 0 before surrounding function returns
for (i = 0; i <= 3; i++) {
  defer fmt.Print(i)
}

// f returns 42
func f() (result int) {
  defer func() {
  // result is accessed after being set to 6 by the return statement
  result *= 7
  }()
  return 6
}
```

`Defer` is commonly used to simplify functions that perform various clean-up actions as seen in the following **`CopyFile`** function:

```
func CopyFile(dstName, srcName string) (written int64, err error) {
  src, err := os.Open(srcName)
  if err != nil {
     return
  }
  defer src.Close()

  dst, err := os.Create(dstName)
  if err != nil {
     return
  }
  defer dst.Close()

  return io.Copy(dst, src)
}
```

`Defer` statements encourage developers to think about closing each file right after opening it, guaranteeing that, regardless of the number of **`return`** statements in the function, the files will be closed.

There are three simple rules governing the behavior of **`defer`** statements:

1. A deferred function's arguments are evaluated when the **`defer`** statement is evaluated.
2. Deferred function calls are executed in last-in-first-out order (LIFO) after the surrounding function returns.

3. Deferred functions may read and assign to the returning function's named return values.

Rule 1 violates C's normal expression evaluation rules because the deferred function's arguments evaluate when the defer statement is evaluated, but the actual function does not.

`Panic` is a built-in function that stops the ordinary flow of control and begins panicking. When the function **`F`** calls **`panic`**, execution of **`F`** stops, any deferred functions in **`F`** are executed normally, and then **`F`** returns to its caller. To the caller, **`F`** then behaves like a call to **`panic`**. The process continues up the stack until all functions in the current goroutine have returned, at which point the program crashes. Panics can be initiated by invoking **`panic`** directly. They can also be caused by runtime errors, such as out-of-bounds array accesses, or arithmetic overflow.

`Recover` is a built-in function that regains control of a panicking goroutine. **`Recover`** is only useful inside deferred functions. During normal execution, a call to recover will return **`nil`** and have no other effect. If the current goroutine is panicking, a call to **`recover`** will return the value that had been given to **`panic`** and then resume normal execution.

The convention in the Go libraries is that even when a package uses **`panic`** internally, its external API still presents explicit error return values.

Other uses of defer include releasing a mutex:

```
mu.Lock()
defer mu.Unlock()
```

printing a footer:

```
printHeader()
defer printFooter()
```

and more.

Appendix F: Examples

This section contains examples of how the defer mechanism may be used in the design of C language programs.

Kernel

Code executing in a kernel environment cannot simply fail or exit and expect the underlying operating system to reclaim system resources for it. As such, code developed for a kernel or similar environments needs to reclaim all allocated resources.

The following example illustrates a kernel function **f1** that acquires two spin locks before accessing shared data:

```
int f1(void) {
  puts("f called");
  if (bad1()) { return 1; }
  spin_lock(&lock1);
  if (bad2()) { goto unlock1; }
  spin_lock(&lock2);
  if (bad()) { goto unlock2; }

  /* Access data protected by the spinlock then force a panic */
  completed += 1;
  unforced(completed);

unlock2:
  spin_unlock(&lock2);
unlock1:
   spin_unlock(&lock1);
   return 0;
}
```

This function can be converted to use the defer mechanism as follows:

```
int f2(void) {
  puts("g called");
  if (bad1()) { return 1; }
  spin_lock(&lock1);
  defer spin_unlock(&lock1);
  if (bad2()) { return 1; }
  spin_lock(&lock2);
  defer spin_unlock(&lock2);
  if (bad()) { return 1; }

  /* Access data protected by the spinlock then force a panic */
  completed += 1;
  unforced(completed);

  return 0;
}
```

This example can be used to illustrate a problem that can occur when modernizing a code base to use the new defer mechanism. For example, assume that function **f1** has not yet been converted to use the defer mechanism and is called as follows:

```
guard {
    defer {
      int err = recover();
      switch (err) {
      case 0:
        puts("no error encountered, continuing as planned");
        break;
      case -1:
        puts("recover and continue...");
        break;
      default:
        printf("unknown error %d, continue\n", err);
        break;
      }
    }
    f1();
  }
```

If the execution of the **unforced** function called from **f1** results in a panic, the function will terminate without releasing the two spinlocks which can result in deadlock when the program attempts to reacquire these locks.

The **unforced** function might panic because of an explicit call to the **panic** function or because an underlying trap condition results in a panic. In cases like this, it may be preferable for the program to crash rather than continue to execute in an unknown state. As a result, it is important that implementations provide a flag which allows compilation units to enable or disable panicking when a trap is detected.

Mutexes

The first example is based on the compliant solution for the CERT C Coding Rule “CON31-C. Do not destroy a mutex while it is locked” [14]. The modified version of the compliant solution is shown below does not use the defer mechanism:

```
mtx_t lock;
unsigned int completed = 0;
enum { max_threads = 5 };

int do_work(void *dummy) {
  if (thrd_success != mtx_lock(&lock)) {
    thrd_exit(thrd_error);
  }
  /* Access data protected by the lock */
  completed += 1;
  if (thrd_success != mtx_unlock(&lock)) {
    exit(EXIT_FAILURE);
  }

  thrd_exit(thrd_success);
}

int main(void) {
  thrd_t threads[max_threads];

  if (thrd_success != mtx_init(&lock, mtx_plain)) {
    exit(EXIT_FAILURE);
  }
  for (size_t i = 0; i < max_threads; i++) {
    if (thrd_success != thrd_create(&threads[i], do_work, NULL)) {
      exit(EXIT_FAILURE);
    }
  }
  for (size_t i = 0; i < max_threads; i++) {
    if (thrd_success != thrd_join(threads[i], 0)) {
      exit(EXIT_FAILURE);
    }
  }

  mtx_destroy(&lock);
  printf("completed = %u.\n", completed);
  return 0;
}
```

The main program starts several threads, each of which locks and increments a counter before returning. The thread identifiers returned by **`thrd_create`** are stored in the **`threads`** array. Most of the errors are treated as terminal failures that invoke **`exit(EXIT_FAILURE)`**. One exception is that the **`do_work`** threads will invoke **`thrd_exit(thrd_error)`** if they cannot create a mutex.

This code can be reimplemented using the defer mechanism as shown below:

```
mtx_t lock;

unsigned int completed = 0;
enum { max_threads = 5 };

int do_work(void *dummy) {
  guard {
    puts("starting do_work guarded block");
    if (thrd_success != mtx_lock(&lock)) {
      thrd_exit(thrd_error);
    }
    defer {
      printf("deferred mtx_unlock = %u.\n", completed);
      if (thrd_success != mtx_unlock(&lock)) {
        panic(thrd_error);
      }
    }
    /* Access data protected by the lock */
    completed += 1;
  }
  thrd_exit(thrd_success);
}

int main(void) {
  guard {
    thrd_t thread;
    int result;
    puts("starting main guarded block");
    if (thrd_success != mtx_init(&lock, mtx_plain)) {
       exit(EXIT_FAILURE);
    }
    defer mtx_destroy(&lock);
    for (unsigned int i = 0; i < max_threads; i++) {
      if (thrd_success != thrd_create(&thread, do_work, NULL)) {
        exit(EXIT_FAILURE);
      }
      defer_capture(thread) {
        if (thrd_success != thrd_join(thread, &result)) {
          printf("thrd_join failure result = %d.\n", result);
          exit(EXIT_FAILURE);
        }
        printf("thread %lu joined result = %d.\n", thread, result);
      }
    } // end for
  } // end guard

  printf("completed = %u.\n", completed);
  return 0;
}
```

The code for this example can be found with the reference implementation https://gitlab.inria.fr/gustedt/defer. In this example, the **`do_work`** thread uses a **`defer`** statement to unlock the mutex. The work of the thread occurs in the guarded block following the defer statement, which is executed when the guarded block exits (for any reason).

The **`main`** function creates and subsequently joins the threads inside a guarded block. The **`defer_capture`** syntax is used to capture the thread identifiers returned by **`thrd_create`** eliminating the need for a local array.

Appendix G: Issues and Limitations

Developer Preference

Many developers strongly prefer the goto-based original to using the defer mechanism. This is expected, as developers who have become familiar with one style of programming are often reluctant to change. Adding a defer mechanism will not prevent or prohibit developers from using existing patterns.

C++

C and C++ are separate languages, but there are ongoing efforts to maintain compatibility between the two language standards including a mailing list for liaison topics spanning WG14 and WG21[9].

C++ typically incorporates the C standard library into the C++ standard library, making some minor alterations along the way. However, the C++ specification doesn't automatically pull in changes to the C language.

In theory, C++ could just ignore the C defer mechanism because they have a better option with exceptions. This would prevent code that needs to be compiled by both C and C++ compilers from using these features.

In cases where C++ code that calls C code that calls C++ code which throws an exception, the exception handling machinery is usually smart enough to handle unwinding through C frames, so C never catches the exception. Either the C++ code catches it, or nothing does. There are some requirements from the C++ that, exceptions thrown from a qsort or bsearch callback must propagate back to the caller of **`qsort`** and **`bsearch`**.[10] There are no further C++ requirements that this works in C, but it's something that implementations commonly support.

[9] https://lists.isocpp.org/mailman/listinfo.cgi/liaison
[10] https://eel.is/c++draft/res.on.exception.handling#2

A similar issue may exist with C code that calls C++ code that calls C code that panics. Implementations that provide the possibility to link C and C++ could provide some levels of compatibility between the two models. For example, they could ensure that C++ destructors are called when C++ code is traversed by a **`panic`**, and possibly translate a **`panic`** with non-zero code to a particular **`stdc_panic_exception`**, say, that behaves similarly to **`recover`**.

There may be overhead to C++ to satisfy the requirements of the **`defer`** feature. A well behaved system would require C++ destructors to be called when a panic crosses C++ code. C++ usually imposes a size overhead on C to satisfy the requirements of C++ exceptions.

Partially Rewritten Code

The defer mechanism can be successful in freeing resources and recovering at higher levels of abstraction if the intervening code has been rewritten to use deferred statements.
Deferred statements have the advantage (and disadvantage) that they can be partially implemented. This allows new code to be written to take advantage of deferred statements while older, existing code continues to use goto chains or other methods to release resources. However, a panic further down in the stack will only release resources that were set to be released using the deferred statements while unwinding the stack. This is not an issue if the panic/recover mechanism is not invoked.

Appendix H: Reference Implementation

A library-only reference implementation of the proposed features [9] is possible, that implements most of the functionality. A reference implementation is distributed with a permissive licence, such that it could easily be integrated by implementations or software projects that want to provide their users with an early preview. The principal restrictions of this reference implementation are its lack of some required diagnostics and a less comfortable handling of some local variables, see below.

In the general case, an implementation can use **`setjmp/longjmp`** to branch to deferred statements. The reference implementation[9] distinguishes jumps to locations known to be within the same function (**`_Defer_shrtjmp`**) from those that are known to jump to another function on the call stack (**`_Defer_longjmp`**). The unwind for a **`return`** will usually be all short jumps, whereas a call to the exit function always initiates a long jump.

The implementation can distinguish cases where all jumps are finally implemented as being long, or platforms where some shortcut for a short jump can be taken. Currently this is only implemented for gcc and friends that implement a "computed **`goto`**", that is labels for which addresses can be taken, and where these addresses then can be used in an extended goto feature.

A computed **goto** is a combination of two new features for C. The first is assigning the addresses of label to an object of type void *.

```
  void* labeladdr = &&somelabel;
somelabel:
   // code
```

The second is invoking **goto** on a variable expression instead of a compile-time-known label, i.e.:

```
void* table[];  // addresses
goto *table[pc];
```

Such a specialized implementation can benefit during unwinding (these then are mostly simple jumps), but they still have to keep track of all the guarded blocks and deferred statements with **setjmp** because these could be jumped to from other functions or from signal handlers. The **setjmp** macro can only appear in a restrictive set of contexts (see Section 7.13.1.1 of the C standard for details). The reference implementation only uses it or its logical negation as the direct controlling expression in an **if** or **for** statement.

The reference implementation has the two different syntaxes for **defer** statements with two different names. The first uses **defer** as proposed here, the second currently uses **defer_capture**. Using **defer_capture** with empty parentheses for the capture has the same effect as the form without capture.

Defer Statements

The reference implementation for gcc and related compilers does not require that **defer** statements be placed within guarded blocks.The frontier between two different calls on the call stack is detected automatically for **return**.

The only special case that is still not covered are functions where there is no return at all (**void** functions or **main**). There we just would fall down the cliff if there is no guard around. The programmer would have to add a **return** statement before the **}** that ends the function body.

This feature uses gcc specifics, namely **__builtin_frame_address(0)**, to obtain and store the frame pointer of a given function call. This is not such a difficult feature and probably most implementations have this, at least conceptually. Using it here on a level of these macros has some performance drawbacks for functions that do not use **defer**:

- all functions are forced to have a frame pointer, no matter what, so hardware register pressure may be higher
- all **return** have a preamble that checks at least one or two pointers dynamically, so tail recursion is probably not well optimized or not optimized at all

Even worse, some inline functions from the C library that have a **return** get a bit shaken. So the **<stddefer.h>** include should always come after all C library includes or even after all other includes if possible. We estimated that for the reference implementation it is important to implement all features correctly.

Local Variables

The values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding **`setjmp`** macro that do not have **`volatile`**-qualified type and have been changed between the **`setjmp`** invocation and **`longjmp`** call are indeterminate (C Standard, Section 7.13.1.1). For our reference implementation, using **`setjmp`**/**`longjmp`** under the hood, this would mean a severe restriction in the usability, because the user would have to carefully qualify some of their local variables. The reference implementation attempts to overcome this difficulty by using synchronization tools from **`<stdatomic.h>`** where these are available.

If these are not available, to allow the use of **`longjmp`** in an implementation, local variables that may change after this defer invocation and that are used in the defer clause must be declared **`volatile`** such that the latest value is accessed.

Volatile qualification is only required for variables that may change between the defer statement and the execution of the deferred statement. For most usages of defer, volatile-qualification should not be necessary. For example a pointer value that you want to free will usually not change once it is allocated:

```
double * x = defer_malloc(sizeof(double[42]));
defer free(x);
```

This code will panic if the allocation fails, and when this doesn't happen, then ensures that the pointed to storage instance is deallocated once the guarded block exits. The pointer should, but is not required to, be const-qualified to emphasize that the pointer value cannot change:

```
double * const x = defer_malloc(sizeof(double[42]));
defer free(x);
```

In addition, to have a determinate value such automatic variables should be defined in a scope that is either surrounding the current guarded block or be that guarded block itself (so they are alive when jumping into the defer clause), or be initialized (by initialization or by assignment) within the defer clause (and so are revived from within).

Appendix I: Stack Unwinding

For objects with automatic storage duration which are not of variable length array type, the lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.

In the C defer mechanism, control moves from a panic to the first **`recover`** statement in a process that is known as *stack unwinding*. Execution of a **`return`**, **`exit`** and **`thrd_exit`** also unwind the stack up to their respective levels of nested guarded blocks.

In stack unwinding, execution proceeds as follows:

1. Control reaches the **`guard`** statement by normal sequential execution. The guarded statements are executed.
2. If no panic occurs during execution of the guarded statements, and no **`return`**, **`exit`**, or **`thrd_exit`** statement is executed, deferred statements are executed in LIFO order and then control continues at the statement immediately following the end of the guarded block.
3. If a panic occurs during execution of the guarded statements or a **`return`** statement, **`exit`**, or **`thrd_exit`** function is executed, all deferred statements registered from the current guarded block are executed in LIFO order.
4. If a **`recover`** function is invoked and returns a value other than zero, the processing of deferred statements stops with the termination of the containing deferred statement, and execution continues as if the guarded block had otherwise terminated normally.
5. After execution of the deferred statements, the guarded block exits and the lifetime of any objects of automatic storage duration declared in the guarded block ends.
6. Execution continues with the execution of any deferred statements in the immediately containing guarded block. The immediately containing guarded block may be in the same function, or in a calling function.
7. Stack unwinding continues until all deferred statements in the function (for return) are executed, thread (for **`panic`** or **`thrd_exit`**) are executed or until all deferred statements in the program (for **`exit`**) are executed.
8. If panicking and no **`recover`** statement is encountered during the whole process, the registered termination function is invoked to terminate the thread or program.
9. Otherwise, execution resumes with the execution of the **`return`** statement , **`exit`**, or **`thrd_exit`** function.

Because the functions **`_Exit`** and **`abort`** are designed for abrupt termination without invoking user defined cleanup mechanisms, neither invokes the defer mechanism to unwind the stack to execute deferred statements. Consequently, invoking either of the functions from within a guarded block may result in resource leaks.

The use of other standard means of nonlinear control flow out of the block (**`goto`**, **`longjmp`**), are constraint violations. For some of these constructs, there are replacements such as **`defer_goto`**, **`defer_abort`** that can be used instead.

Appendix J: Interaction with Signals

To ensure a consistent resource management and security, it is important that the defer mechanism is successfully embedded into the general failure model of the implementation. Therefore, a well-defined interaction with signal handling is encouraged.

The current specification in the C standard for signals is rudimentary, and so we cannot provide a feature-complete proposal for an interaction with the signal subsystem. But we can request that implementations document the interaction between signals and the defer mechanism and we can also recommend some practice. We propose to add the following two paragraphs as 7.14 p5 and p6.

> It is implementation-defined if default signal handlers that terminate the execution previously execute deferred statements of the current or any other thread and if the processing of deferred statements is stopped by a call to **`recover`**.
>
> **Recommended practice**
>
> It is recommended that default signal handlers other than for **`SIGABRT`** that terminate the execution do so as if by calling **`panic`**, **`exit`**, **`quick_exit`** or **`_Exit`** and that they execute the pending deferred statements and **`atexit`** or **`at_quick_exit`** handlers as described for these functions.

For strictly conforming programs, only the C standard library functions **`abort`**, **`_Exit`**, **`quick_exit`**, and **`signal`** can be safely called from within a signal handler. This proposal adds the **`panic`** macro to that list.

When a panic is triggered by a signal handler (either a default handler or user defined) the execution of the deferred statements themselves is performed in the context where they originally appeared, not in the context of the signal handler. Thus restrictions for signal handlers do not apply to the deferred statements themselves, only that information that a signal handler wants to communicate to the deferred statements must either be the error code to panic (which then can be recovered) or by using objects that have a lock-free atomic type or are a **`volatile`** qualified **`sig_atomic_t`**.

Appendix J: Performance Considerations

The zero-overhead principle It has been repeatedly stated that C++ exceptions violate the zero-overhead principle. Unsurprisingly, the answer depends on how you interpret the zero-overhead principle. Zero-overhead is not zero-cost; it is zero-overhead compared to roughly equivalent functionality. Similarly, the exception mechanism was compared to alternatives in a program that needed to catch all errors and then either recover or terminate.

Zero cost is more applicable to features that are not being used contributing to the cost of an operation. Obviously there can be no such thing as a zero-cost unwinding because the unwinding is an action. But this is also much more important for RAII than it would be for deferred execution where potentially any old object (and thousands of them) need to take an action on unwinding. Release of **`unique_ptr`** would be the obvious example of something you probably don't want to silently spend time establishing **`setjmp`** frames for during non-exceptional use. But this seems like it should be less of a problem for C because there seem to be several implications of **`defer`**:

- it's explicit, which means the user can see that they may need to expect some setup/teardown around the guarded block, and also not to expect it elsewhere
- it would likely represent a high-cost but low-intensity operation (and QoI could allow a compiler to point out that putting **`defer`** inside a loop body isn't necessarily a good idea); but if it is used that way, it's still explicit
- it's only likely to be used where the exceptional case is, even if not necessarily a likely path, locally important; it doesn't affect code that isn't at least interested in the exceptional case
- the actual operations associated with setup/teardown are likely going to be far more expensive than the frame handling for C-like use cases

The **`Boost.ScopeGuard`** library gives a good idea how defer could be at least partially implemented in C++. **`Boost.ScopeGuard`** tends to optimize well. It's zero-cost in the sense that the compiler fully inlines the argument. Lookup cost is only incurred on the **`throw`**-path. This is an important implementation experience that shows that compilers should be able to do a good job with statically analyzable **`defer`** uses in practice once they start to implement it as a primitive.

While this is true for a facility like **`scope_guard`** that introduces a static scope at every instantiation, the proposed defer facility is dynamic. That dynamism requires it to be invoked through indirection in at least some cases and potentially unbounded allocations (much like go's defer). The optimization would be possible for this kind of defer in simple functions with no interesting control flow, but not in the general case.

The unconditional association with a statically-known scope is going to be the common case by far. Codebases that make use of related features have found that deferred execution costs are inlined away all the time in practice.

Because of the general case, the library implementation would find it hard to optimize, but a native compiler-provided feature would know when it's the general case vs. the common case and be able to provide different backing implementations accordingly. This strengthens the case for encouraging implementers to provide at least a surface level of native support. Implementations could, for instance, detect when the use is "RAII-like" and translate to the C++-like IR, and when it isn't, just use the reference implementation this gives us a really nice middle ground of low-effort/high-reward for implementers.

Deferred statements are comparable to Objective-C and @autorelease - although that's a "purely" memory-related feature (it frees, it doesn't do other things). Autorelease pools can be dynamically-stacked, which may provide some interesting implementation experience with respect to the general case and optimizatibility for **`defer`** / **`guard`** because some autorelease pools can be statically associated with an allocation and many others cannot.

The main go compiler began to do this same thing relatively recently (https://go.googlesource.com/proposal/+/refs/heads/master/design/34481-opencoded-defers.md) Auto release is a good corollary, and one implemented and used in a custom defer mechanism, also relates well to the library support in talloc. The problem case is the **`defer`** table used in a **`for`** loop necessarily must be dynamically sized unless you have compile time loop bounds. That pattern is a pretty common antipattern in go, and an easy way for this to go wrong.

Appendix K: Modifications to Existing Features

This section contains changes to existing features of the C language.

The **`break`** statement

A **`break`** statement can be used to terminate a **`guard`** or **`defer`** statement by acting as a goto to a label immediately before the closing brace of the guard statement. All deferred statements contained in the guarded statements are executed just before the guarded block exits.

Constraints

A **`break`** statement shall appear only in or as a **`switch`** body, loop body, deferred statement, or guarded block.

Semantics

A **`break`** statement terminates execution of the smallest enclosing switch, iteration, deferred, or guarded statement.

EXAMPLE 1 Exit a guarded statement using a **`break`** statement as the result of an unrecoverable error:

```
guard {
   double* A = malloc(sizeof(double[n]));
   defer free(A);
   /* do something complicated */
   if (something_went_wrong) break;
   /* do something even more complicated */
   if (something_else_went_wrong) break;
   /* do something very complicated */
}
```

EXAMPLE 2 The deferred **`break`** statement in the following code segment serves no purpose and can be omitted without changing the behavior of the program:

```
guard {
   defer puts("executed deferred statement");
   defer break; // no-op
}
```

Rationale

Changing the semantics of the **`break`** statement to break out of a guarded statement only makes sense if we use the **`guard`** keyword. If the **`guard`** keyword is eliminated, this change will also be eliminated.

The following code segment does not use defer:

```
for (int i = 0; i < 10; ++i) {
  void *ptr = malloc(12);
  if (!ptr) break;
  ...
  free(ptr);
}
```

A programmer might incorrectly introduce defer as follows:

```
for (int i = 0; i < 10; ++i) guard {
  void *ptr = malloc(12);
  if (!ptr) break; // Does not break out of the for loop
  defer free(ptr);
  ...
}
```

Instead the code would need to be restructured to ensure that the code breaks out of the **`for`** loop, for example:

```
for (int i = 0, bail_out = 0; !bail_out && i < 10; ++i) guard {
  void *ptr = malloc(12);
  if (!ptr) {
    bail_out = 1; // Next iteration of for loop doesn't run
    break; // Breaks out of the guard block
  }
  defer free(ptr);
  ...
}
```

Terminating functions

The `exit` function

The beginning of 7.22.4.4 p 3 is changed as follows:

> First, all deferred statements of all active function calls of the current thread, if any, are sequenced in the inverse order in which they have been met during execution. Then, all functions registered by the **`atexit`** function are called, in the reverse order of their registration …

The `thrd_exit` function

The beginning of 7.26.5.5 p 2 is changed as follows:

> First, all deferred statements of all active function calls of the current thread, if any, are sequenced in the inverse order in which they have been met during execution. Then, for every thread-specific storage key …

Rationale

The **`quick_exit`** function was adapted from C++ where its primary purpose concerned with was to avoid executing static destructors in C++ in situations where cooperative cancellation is not possible [17]:

"The C++ Committee has clearly stated that it wishes to preserve execution of static destructors in normal applications. So, we need a mechanism to abandon an application process without cooperatively canceling all threads and without executing the static destructors."

Given this history, our current view is that the **`quick_exit`** function will not be modified to evaluate deferred statements.

C Library Augmentations

As examples of how panic could be used by the C library, there could be augmented versions of storage allocation functions, such as **`defer_malloc`**. The idea is that these will panic if they encounter an out-of-memory condition. That has several effects:

- Explicit handling of such error conditions does not need to be repeated at every call site.
- Cleanup actions that the user has installed by means of defer clause will be called, e.g., to close files or to free large allocations.
- User code may establish a recovery mechanism using recover. This will probably be rarely used by most applications, but safety/security critical applications would get a handle to avoid catastrophes.

Inspection of the assembler that is created for these functions shows that all of this generates little overhead for the fast execution path. In general, this technique might provide a sensible way to provide the same error detection facilities as Annex K, but in a way that:

- preserves the prototypes of the functions
- is thread safe.

Appendix J: Panic Handler

Defer `<stddefer.h>`

The header `<stddefer.h>` defines one type.
The type is

```
    panic_handler_t
```

which has the following definition

```
    typedef void (*panic_handler_t)(int code);
```

The `set_panic_handler` function

Synopsis

```
    #include <stddefer.h>
    panic_handler_t set_panic_handler(panic_handler_t handler);
```

Description

The **`set_panic_handler`** function sets the panic handler for the current thread to **`handler`**. The panic handler is the function to be called to terminate the current thread or program following the execution of all deferred statements as the result of an unrecovered panic.

When the handler is called, it is passed the **`code`** set by the call to the **`panic`** function.

The implementation has a default panic handler that is used if no calls to the **`set_panic_handler`** function have been made. The behavior of the default handler is implementation-defined. Recommended practice is to use either the **`exit`** or **`quick_exit`** functions.

If the handler argument to **`set_panic_handler`** is a null pointer, the implementation default handler becomes the current panic handler.

The **`set_panic_handler`** is thread safe, and cannot be changed from another thread.

Only the most recent handler registered with **`set_panic_handler`** is called when an unrecovered panic occurs. The registered panic handler has thread storage duration. The registered panic handler for a newly created thread shall be the same as the registered panic handler of the current thread at the time of creation.

Returns

The **`set_panic_handler`** function returns a pointer to the previously registered handler.[11]

Rationale

The **`set_panic_handler`** function is akin to the **`set_terminate`** in C++ that establishes the current handler function for terminating exception processing. In C++, setting the termination handler local to a thread can be cumbersome for safety critical applications that want to put the application into a safe state if something bad happens, no matter what thread it happens on.

---

[11] If the previous handler was registered by calling **`set_panic_handler`** with a null pointer argument, a pointer to the implementation default handler is returned (not **`NULL`**).

This may be a sufficient reason for this function to apply to the program and not the thread, or to parameterize the function.

References

[1]	Understanding Denial-of-Service Attacks | CISA
[2]	CWE - Common Weakness Enumeration
[3]	CWE - CWE-400: Uncontrolled Resource Consumption (4.1)
[4]	CWE - 2019 CWE Top 25 Most Dangerous Software Errors
[5]	CWE - CWE-415: Double Free (4.1)
[6]	Why mobile web apps are slow
[7]	WG14 N1841 - Alternate Exception Handling Syntax for C
[8]	https://ziglang.org/#A-fresh-take-on-error-handling
[9]	https://gitlab.inria.fr/gustedt/defer
[10]	James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman. The Java® Language Specification, Java SE 14 Edition. 2020-02-20 https://docs.oracle.com/javase/specs/jls/se14/html/jls-14.html#jls-14.20.3
[11]	Abrahams D. (2000) Exception-Safety in Generic Components. In: Jazayeri M., Loos R.G.K., Musser D.R. (eds) Generic Programming. Lecture Notes in Computer Science, vol 1766. Springer, Berlin, Heidelberg
[12]	Paul Ducklin, Anatomy of a “goto fail” – Apple’s SSL bug explained, plus an unofficial patch for OS X!. Feb 2014.

	https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/
[13]	Lawrence Crowl. N1483 Comparing Lambda in C Proposal N1451 and C++ FCD N3092 2010-05-29 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1483.htm
[14]	Robert C. Seacord. 2014. The CERT® C Coding Standard, Second Edition: 98 Rules for Developing Safe, Reliable, and Secure Systems (2nd. ed.). Addison-Wesley Professional.
[15]	Niall Douglas. P1095R0/N2289: Zero overhead deterministic failure A unified mechanism for C and C++. 2018-08-29. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2289.pdf
[16]	Eli Bendersky. Handling out-of-memory conditions in C. October 30, 2009. https://eli.thegreenplace.net/2009/10/30/handling-out-of-memory-conditions-in-c
[17]	P.J. Plauger. WG14 N1327 Abandoning a Process 05-Aug-2008 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1327.htm