DRAFT Technical Specification

ISO/DIS TS 25755

Programming Languages — C — defer, a mechanism for general purpose, lexical scope-based undo

ISO/ TC22/SC22
Secretariat: JISC

Voting begins on: n/a

Voting terminates on: n/a

Reference Number
ISO/DIS TS 25755 : Working Draft N3928
Project Editor: JeanHeyd Meneide (wg14@soasis.org)

⚠ COPYRIGHT PROTECTED DOCUMENT

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Content

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming languages, their environments and system software interfaces.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

The advent of resource leaks in programs created with ISO/IEC 9899 —— Programming Languages, C has necessitated the need for better ways of tracking and automatically releasing resources in a given scope. This document provides a feature to address this need in a reliable, translation-time, opt-in manner for implementations to furnish to programmers.

This document is divided into four major subdivisions:

— preliminary elements (Clauses 1-4);
— the characteristics of environments that translate and execute C programs (Clause 5);
— the language syntax, constraints, and semantics (Clause 6);
— the library facilities (Clause 7).

In any given subsequent clause or subclause, there are section delineations in bold to describe the semantics, restrictions, and behaviors of programs for this language and potentially the use of its library clauses in this document:

— **Syntax**
which pertains to the spelling and organization of the language and library;
— **Constraints**
which detail and enumerate various requirements for the correct interpretation of the language and library, typically during translation;
— **Semantics**
which explain the behavior of language features and similar constructs;
— **Description**
which explain the behavior of library usage and similar constructs;
— **Returns**
which describes the effects of constructs provided back to a user of the library;
— **Recommended practice**
which provides guidance and important considerations for implementers of this document.

Examples are provided to illustrate possible forms of the constructions described. Footnotes are provided to emphasize consequences of the rules described in that subclause or elsewhere in this document. References are used to refer to other related subclauses. Recommendations are provided to give advice or guidance to implementers.

References internal to claues and subclauses in this document are specified by a stable tag that beings with a “[“, followed by a list of letters, numbers, hyphens, and periods, and is terminated with a final “]”. There is no structure or hierarchy to the name between the left and right square brackets of the stable tags. Clauses and subclauses in other documents uses section numbers or other appropriate reference nomenclature.

1. Scope [scope]

1 This Technical Specification specifies a series of extensions of the programming language C, specified by the international standard ISO/IEC 9899:2024.

2 Each clause in this Technical Specification deals with a specific topic. The first sub-clauses of clauses 4 through 7 contain a technical description of the features of the topic and what is necessary for an implementation to achieve conformance through extensions or additions to ISO/IEC 9899:2024.

2. Normative References [normrefs]

1 The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 9899:2024, Programming languages — C

3. Terms and definitions [defn]

1 For the purposes of this document, the terms and definitions of ISO/IEC 9899:2024 apply.

4. Conformance [conf]

1 The requirements from ISO/IEC 9899:2024, clause 4 apply without any additional requirements in this document.

5. Environment [env]

5.1. General [env-general]

1 The requirements from ISO/IEC 9899:2024, clause 5 apply along with the following additional requirements to support the defer feature.

5.2. Program termination [prog.term]

Semantics

1 If the return type of the `main` function is a type compatible with **`int`**, a **`return`** from the initial call to the main function is equivalent to calling the `exit` function with the value returned by the `main` function as its argument after all defer statements that are in scope for the `main` function have been executed.

6. Language [lang]

6.1. General [lang-general]

1 The requirements from ISO/IEC 9899:2024, clause 6 apply along with the following additional requirements to support the **`_Defer`** feature.

6.2. Keywords [keywords]

1 In addition to the keywords in ISO/IEC 9899:2024 §6.4.2, an implementation shall additionally recognize **`_Defer`** as a keyword.

Recommended practice

2 Implementations are encouraged to provide an additional keyword **`defer`** as an alternative spelling for the **`_Defer`** keyword. It should have all the significance of the **`_Defer`** keyword described in this document.

6.3. Statements [statements]

1 In addition to the statements in ISO/IEC 9899:2024 §6.8, implementations shall allow the unlabeled statement grammar production to produce a defer statement which contains a deferred block. A deferred block is also considered a *block* just like a primary block or a secondary block.

Syntax

2 *unlabeled-statement*:

expression-statement
attribute-specifier-sequence$_{\text{opt}}$ *primary-block*
attribute-specifier-sequence$_{\text{opt}}$ *jump-statement*
attribute-specifier-sequence$_{\text{opt}}$ *defer-statement*

deferred-block:

unlabeled-statement

6.4. Defer statements [defer]

Syntax

1 *defer-statement*:

`_Defer` *deferred-block*

Description

2 Let *D* be a defer statement, *S* be the deferred block of *D*, and *E* be the enclosing block of *D*. The scope of *D* is the same as an identifier declared and completed immediately after the end of *S*.

Constraints

3 Jumps by means of:

- **`goto`** or **`switch`** shall not jump into any defer statement;
- **`goto`** or **`switch`** shall not jump from outside the scope of a defer statement *D* to inside that scope;
- and, **`goto`**, **`break`**, **`continue`** or **`return`** shall not exit *S*.

Semantics

4 When execution reaches a defer statement *D* and its scope is entered, its *S* is not immediately executed during sequential execution of the program. Instead, for the duration of the scope of *D*, *S* is executed upon:

- the termination of the block *E* and/or the scope of *D* (such as from reaching its end);
- or, any exit from *E* and/or the scope of *D* through **`return`**, **`goto`**, **`break`**, or **`continue`**.

The execution is done just before leaving the enclosing block *E* and/or the scope of *D*. In particular **`return`** expressions (and conversion to return values) are calculated before executing *S*.

5 Multiple defer statements execute their *S* in the reverse order they appeared in *E*. Within a single defer statement *D*, if *D* contains one or more defer statements D_{sub} of its own, then the S_{sub} of the D_{sub} are also executed in reverse order at the termination and/or exit of E_{sub} and/or D_{sub}'s scope, recursively, according to the rules of this subclause.

6 If a non-local jump (e.g. `longjmp`) is used in *D*'s scope but before the execution of the *S* of *D*:

— if execution leaves *D*'s scope, *S* is not executed;
— otherwise, if control returns to a point in *E* and causes *D* to be reached more than once, the effect is the same as reaching *D* only once.

7 NOTE 1 The "execution" of a defer statement only ensures that *S* is run on any exit from *D*'s scope. There is no observable side effect to repeat from reaching *D*, as the manifestation of any of the effects of *S* happen if and only if the scope of *D* is exited or terminated after reaching *D*, as previously specified. "Tracking" of reached defer statements at execution time is not necessary: if the non-local jump leaves the scope it is not executed (forgotten); and, if it is reached again it behaves as it would during normal execution.

8 If a non-local jump (e.g. `longjmp`) is executed from *S* and control leaves *S*, the behavior is undefined.

9 If a non-local jump (e.g. `longjmp`) is executed outside of any *D* and:

— it jumps into any *S*;
— or, it jumps outside any *D*'s scope to inside that *D*'s scope;

the behavior is undefined.

10 If *E* has any defer statements *D* that have been reached and their *S* have not yet executed, but the program is terminated or leaves the scope of *D* through any means not specified previously, including but not limited to:

— a function with the `_Noreturn` function specifier, or a function annotated with the `noreturn` or `_Noreturn` attribute, is called;
— or, any signal `SIGABRT`, `SIGINT`, or `SIGTERM` occurs;

then any such *S* are not run. Any other *D* that have not been reached do not have their *S* run.

11 EXAMPLE 1 Defer statements cannot be jumped over.

```
#include <stdio.h>

int f() {
     goto target; // constraint violation
     _Defer { fputs(" meow\n", stdout); }
target:
     fputs("cat says", stdout);
     return 1;
}

int g() {
     return fputs("cat says", stdout);
     _Defer { fputs(" meow\n", stdout); } // okay: no constraint violation,
     // not executed
     // print "cat says" to standard output
```

```
}

int h() {
     goto target;
     {
          // okay: no constraint violation
          _Defer { fputs(" meow\n", stdout); }
     }
target:
     fputs("cat says", stdout);
     return 1; // prints "cat says" to standard output
}

int i() {
     {
          _Defer { fputs("cat says", stdout); }
          // okay: no constraint violation
          goto target;
     }
target:
     fputs(" meow\n", stdout);
     return 1; // prints "cat says meow" to standard output
}

int j() {
     _Defer {
          goto target; // constraint violation
          fputs(" meow\n", stdout);
     }
target:
     fputs("cat says", stdout);
     return 1;
}

int k() {
     _Defer {
          return 5; // constraint violation
          fputs(" meow\n", stdout);
     }
     fputs("cat says", stdout);
     return 1;
}

int l() {
     _Defer {
     target:
          fputs(" meow\n", stdout);
     }
     goto target; // constraint violation
     fputs("cat says", stdout);
     return 1;
}

int m() {
     goto target; // okay: no constraint violation
```

```
     {
     target:
          _Defer { fputs("cat says", stdout); }
     }
     fputs(" meow\n", stdout);
     return 1; // prints "cat says meow" to standard output
}

int n() {
     goto target; // constraint violation
     {
          _Defer { fputs(" meow\n", stdout); }
     target:
     }
     fputs("cat says", stdout);
     return 1;
}

int o() {
     {
          _Defer fputs("cat says", stdout);
          goto target;
     }
target:;
     fputs(" meow\n", stdout);
     return 1; // prints "cat says meow"
}

int p() {
     {
          goto target;
          _Defer fputs(" meow\n", stdout);
     }
target:;
     fputs("cat says", stdout);
     return 1; // prints "cat says"
}

int q() {
     {
          _Defer { fputs(" meow\n", stdout); }
     target:
     }
     goto target; // constraint violation
     fputs("cat says", stdout);
     return 1;
}

int r() {
     {
     target:
          _Defer { fputs("cat says", stdout); }
     }
     goto target; // ok: prints "cat says" repeatedly
     // never reached
```

```
     fputs(" meow\n", stdout);
     return 1;
}

int s() {
     {
     target:
          _Defer { fputs("cat says", stdout); }
          goto target; // ok
          // prints "cat says" repeatedly
     }
     // never reached
     fputs(" meow", stdout);
     return 1;
}

int t() {
     int count = 0;
     {
     target:
          _Defer { fputs("cat says ", stdout); }
          ++count;
          if (count <= 2) {
               goto target; // ok
          }
     }
     fputs("meow", stdout);
     return 1; // prints "cat says cat says cat says meow"
}

int u() {
     int count = 0;
     {
          _Defer { fputs("cat says", stdout); }
     target:
          if (count < 5) {
               ++count;
               goto target; // ok
          }
     }
     fputs(" meow", stdout);
     return 1; // prints "cat says meow"
}

int v() {
     int count = 0;
target:
     if (count >= 2) {
          fputs("meow", stdout);
          return 1; // prints "cat says cat says meow"
     }
     _Defer { fputs("cat says ", stdout); }
     count++;
     goto target;
```

```
     return 0; // never reached
}
```

12 EXAMPLE 2 All the expressions and statements of an enclosing block are evaluated before executing defer statements, including any conversions. After all defer statements are executed, the block is then exited.

```
int main() {
     int r = 4;
     int* p = &r;
     _Defer { *p = 5; }
     return *p; // return 4;
}
```

This is important for proper resource management in conjunction with potentially complex return expressions.

```
#include <stdlib.h>
#include <stddef.h>

int use_buffer(size_t n, void* buf) {
     /* ... */
     return 0;
}

int main() {
     const int size = 20;
     void* buf = malloc(size);
     _Defer { free(buf); }
     // buffer is not freed until AFTER use_buffer returns
     return use_buffer(size, buf);
}
```

Conversions for the purposes of **return** are also computed before any defer statements are entered.

```
#include <float.h>
#include <fenv.h>
#include <assert.h>

bool f() {
     double x = DBL_SNAN;
     _Defer {
#pragma STDC FENV_ACCESS ON
          // fetestexcept(FE_INVALID) is nonzero because of the
          // comparison during the conversion to bool
          assert(fetestexcept(FE_INVALID) != 0);
     }
     return x;
}
```

13 EXAMPLE 3 It is not defined if defer statements execute their deferred blocks if the exiting / non-returning functions detailed previously are called.

```
#include <stdlib.h>

int f() {
     void* p = malloc(1);
     if (p == NULL) {
          return 0;
     }
     _Defer free(p);
     exit(1); // "p" may be leaked
     return 1;
}

int main() {
     return f();
}
```

14 EXAMPLE 4 Defer statements, when execution reaches them, are tied to the scope of the defer statement within their enclosing block, even if it is a secondary block without braces.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
     {
          _Defer {
               fputs(" meow", stdout);
          }
          if (true)
               _Defer fputs("cat", stdout);
          fputs(" says", stdout);
     }
     // "cat says meow" is printed to standard output
     exit(0);
}
```

15 This applies to any enclosing block, even **for** loops without braces around its body.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
     const char* arr[] = {"cat", "kitty", "ferocious little baby"};
     _Defer {
          fputs(" meow", stdout);
     }
     for (unsigned int i = 0; i < 3; ++i)
          _Defer printf("my %s,\n", arr[i]);
     fputs("says", stdout);

     // "my cat,
     // my kitty,
     // my ferocious little baby,
     // says meow"
```

```
     // is printed to standard output
     return 0;
}
```

16 EXAMPLE 5 Defer statements execute their deferred blocks in reverse order of the appearance of the defer statements, and nested defer statements execute their deferred blocks in reverse order but at the end of the deferred block they were invoked within. The following program:

```
int main() {
     int r = 0;
     {
          _Defer {
               _Defer r *= 4;
               r *= 2;
               _Defer {
                    r += 3;
               }
          }
          _Defer r += 1;
     }
     return r; // return 20;
}
```

is equivalent to:

```
int main() {
     int r = 0;
     r += 1;
     r *= 2;
     r += 3;
     r *= 4;
     return r; // return 20;
}
```

17 EXAMPLE 6 Defer statements can be executed within a **switch**, but a **switch** cannot be used to jump into the scope of a defer statement.

```
#include <stdlib.h>

int main() {
     void* p = malloc(1);
     switch (1) {
     _Defer free(p); // constraint violation
     default:
          _Defer free(p);
          break;
     }
     return 0;
}
```

18 EXAMPLE 7 Defer statements can not be exited by means of **break** or **continue** .

```
int main() {
     switch (1) {
     default:
          _Defer {
               break; // constraint violation
          }
     }
     for (;;) {
          _Defer {
               break; // constraint violation
          }
     }
     for (;;) {
          _Defer {
               continue; // constraint violation
          }
     }
     return 0;
}
```

19 EXAMPLE 8 Defer statements that are not reached are not executed.

```
#include <stdlib.h>

int main() {
     void* p = malloc(1);
     return 0;
     _Defer free(p); // not executed, p is leaked
}
```

20 EXAMPLE 9 Defer statements can contain other compound statements.

```
typedef struct meow *handle;

extern int purr(handle *h);
extern void un_purr(handle h);

int main() {
     handle h;
     int err = purr(&h);
     _Defer if (!err) un_purr(h);
     return 0;
}
```

21 EXAMPLE 10 Non-local jumps do not execute defer statements, and jumping backwards to reach the same defer statement does not accumulate defer statements.

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

static jmp_buf jbuf;
```

```
int f(int count) {
     _Defer { fputs("cat says meow", stdout); }
     if (count == 2) {
          return 1;
     }
     longjmp(jbuf, count + 1);
     return 0;
}

int main(void) {
     volatile int count = 0;
     if (setjmp(jbuf) == 2) {
          // prints "cat says meow"
          return 1;
     }
     f(count);
     return 0;
}
```

6.5. Predefined macro names [predef.macro]

1 In addition to the keywords in ISO/IEC 9899:2024 §6.10.10, an implementation shall define the following macro names:

`__STDC_DEFER_TS25755__` The integer literal `2` if the implementation follows the recommended practice and provides a keyword **`defer`** as in **[keywords]**. Otherwise, the integer literal `1`.

7. Library [library]

1 The requirements from ISO/IEC 9899:2024, clause 7 apply with additional requirements in this document.

7.1. The `thrd_create` function [lib-thrd.create]

1 In addition to the description and return requirements in in ISO/IEC 9899:2024 §7.28.5.1, when the `thrd_start_t func` parameter is returned from, it behaves as if it also runs any defer statements that are in scope for `func` before invoking `thrd_exit` with the returned value.

7.2. Defer mechanism `<stddefer.h>` [lib-stddefer.hdr]

1 The header `<stddefer.h>` defines the following macros for use with the defer feature.

2 The macro

```
__STDC_VERSION_STDDEFER_H__
```

is an integer constant expression with the value `202602L`.

3 The macro

```
defer
```

which expands to **`_Defer`**.

Index