DRAFT Technical Specification

ISO/DIS TS XYZW

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 XYZW : Working Draft N3328

⚠ 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, static, 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.

1 Scope

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

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 modifications or additions to ISO/IEC 9899:2024.

2 Normative References

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

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

4 Conformance

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

5 Environment

5.1 General

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

Semantics

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 active defer statements of the function body of main have been executed.

6 Language

6.1 General

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

6.2 Keywords

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

6.3 Statements

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.

Syntax

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

6.4 Defer statements

Syntax

defer-statement:
defer *secondary-block*

Description

Let *D* be a defer statement, *S* be the secondary block of *D* referred to as its deferred content, and *E* be the enclosing block of *D*.

Constraints

Jumps by means of **goto** or **switch** shall not jump into any defer statement.

Jumps by means of **goto** or **switch** into *E* shall not jump over a defer statement in *E*.

Jumps by means of **goto** in *E* shall not jump over a defer statement in *E*.

Jumps by means of **return** shall not exit *S*.

Semantics

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

— the termination of the block *E* (such as from reaching its end);
— or, any exit from *E* through **return**, **goto**, **break**, or **continue**.

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

Multiple defer statements execute in the reverse lexical order they appeared in *E*. Within a single defer statement *D*, if *D* contains one or more defer statements of its own, then these defer statements are also executed in reverse lexical order at the end of *S*, recursively, according to the rules of this clause.

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 *E* through any means including:

— a function with the deprecated `_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, unless otherwise specified by the implementation. Any other *D* that have not been reached are not run.

NOTE 1 The execution of deferred statements upon non-local jumps (i.e., `longjmp` and `setjmp` described in ISO/IEC 9899:2024 §7.13) or program termination is a technique sometimes known as

"unwinding" or "stack unwinding", and some implementations perform it. See also ISO/IEC 14882 Programming languages — C++ [except.ctor].

If a non-local jump is used within *E* but before the execution of *D*:

— if execution leaves *E*, *S* will not be executed;
— otherwise, if control returns to a point in *E* and causes *D* to be reached more than once, there is no effect.

NOTE 2 The "execution" of a defer statement only lets the program know that *S* will be run on any exit from that scope. There is no observable side effect to repeat from reaching *D*, as the manifestation of any of the effects of *S* will happen if and only if *E* is exited or terminated as previously specified.

If a non-local jump is executed from *S* and control leaves *S*, the behavior is unspecified.

If a non-local jump is executed outside of any *D* and:

— it jumps into any *S*;
— or, it jumps over any *D*;

the behavior is unspecified.

EXAMPLE 1 Defer statements cannot be jumped over.

```
#include <stdio.h>

int f () {
     goto b; // constraint violation
     defer { printf(" meow"); }
     b:
     printf("cat says");
     return 1;
}

int g () {
     return printf("cat says");
     defer { printf(" meow"); } // okay: no constraint violation, not executed
     // print "cat says" to standard output
}

int h () {
     goto b;
     {
          // okay: no constraint violation
          defer { printf(" meow"); }
     }
     b:
     printf("cat says");
     return 1; // prints "cat says" to standard output
}

int i () {
     {
          defer { printf("cat says"); }
          // okay: no constraint violation
          goto b;
     }
```

```
     b:
     printf(" meow");
     return 1; // prints "cat says meow" to standard output
}

int j () {
     defer {
          goto b; // okay: no constraint violation
          printf(" meow");
     }
     b:
     printf("cat says");
     return 1; // prints "cat says" over
     // and over again to standard output
}

int k () {
     defer {
          return 5; // constraint violation
          printf(" meow");
     }
     printf("cat says");
     return 1;
}

int l () {
     defer {
          b:
          printf(" meow");
     }
     goto b; // constraint violation
     printf("cat says");
     return 1;
}

int m () {
     goto b; // okay: no constraint violation
     {
          b:
          defer { printf("cat says"); }
     }
     printf(" meow");
     return 1; // prints "cat says meow" to standard output
}

int n () {
     goto b; // constraint violation
     {
          defer { printf(" meow"); }
          b:
     }
     printf("cat says");
     return 1;
}
```

```
int o () {
     {
          defer printf("cat says");
          goto b;
     }
     b:;
     printf(" meow");
     return 1; // prints "cat says meow"
}

int p () {
     {
          goto b;
          defer printf(" meow");
     }
     b:;
     printf("cat says");
     return 1; // prints "cat says"
}
```

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;
}
```

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

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

bool f () {
     double x = DBL_SNAN;
     defer {
          // fetestexcept (FE_INVALID) is nonzero because of the
          // comparison during the conversion to bool
          assert(ftestexcept(FE_INVALID) != 0);
     }
     return x;
}
```

EXAMPLE 3 It is implementation-defined if defer statements will execute if the exiting / non-returning functions detailed previously are called.

```
#include <stdio.h>
#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();
}
```

EXAMPLE 4 Defer statements, when execution reaches them, are tied to their enclosing block.

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

int main () {
     {
          defer {
               printf(" meow");
          }
          if (true)
               defer printf("cat");
          printf(" says");
     }
     // "cat says meow" is printed to standard output
     exit(0);
}
```

EXAMPLE 5 Defer statements execute in reverse lexical order, and nested defer statements execute in reverse lexical order but at the end of the defer statement 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;
}
```

EXAMPLE 6 Defer statements can be executed within a **switch**, but a switch cannot be used to jump over 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 2;
}
```

EXAMPLE 7 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
}
```

EXAMPLE 8 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;
}
```

6.5 Predefined macro names

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

__STDC_DEFER_TS___ The integer literal 1.

7 Library

The requirements from ISO/IEC 9899:2024, clause 7 apply without any additional requirements in this document.

Index