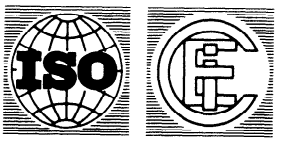
INTERNATIONAL STANDARD

ISO/IEC 9899

First edition
1990-12-15

Programming languages — C

Langages de programmation — C



Reference number
ISO/IEC 9899 : 1990 (E)

Contents

Annexes

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.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 9899 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

Annexes A, B, C, D, E, F and G are for information only.

Introduction

With the introduction of new devices and extended character sets, new features may be added to this International Standard. Subclauses in the language and library clauses warn implementors and programmers of usages which, though valid in themselves, may conflict with future additions.

Certain features are *obsolescent*, which means that they may be considered for withdrawal in future revisions of this International Standard. They are retained because of their widespread use, but their use in new implementations (for implementation features) or new programs (for language [6.9] or library features [7.13]) is discouraged.

This International Standard is divided into four major subdivisions:

— the introduction and preliminary elements;

— the characteristics of environments that translate and execute C programs;

— the language syntax, constraints, and semantics;

— the library facilities.

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 International Standard. References are used to refer to other related subclauses. A set of annexes summarizes information contained in this International Standard. The introduction, the examples, the footnotes, the references, and the annexes are not part of this International Standard.

The language clause (clause 7) is derived from ''The C Reference Manual'' (see annex A).

The library clause (clause 8) is based on the *1984 /usr/group Standard* (see annex A).

This page intentionally left blank

Programming languages — C

1 Scope

This International Standard specifies the form and establishes the interpretation of programs written in the C programming language.[1] It specifies

— the representation of C programs;

— the syntax and constraints of the C language;

— the semantic rules for interpreting C programs;

— the representation of input data to be processed by C programs;

— the representation of output data produced by C programs;

— the restrictions and limits imposed by a conforming implementation of C.

This International Standard does not specify

— the mechanism by which C programs are transformed for use by a data-processing system;

— the mechanism by which C programs are invoked for use by a data-processing system;

— the mechanism by which input data are transformed for use by a C program;

— the mechanism by which output data are transformed after being produced by a C program;

— the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor;

— all minimal requirements of a data-processing system that is capable of supporting a conforming implementation.

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 646:1983, *Information processing — ISO 7-bit coded character set for information interchange*.

ISO 4217:1987, *Codes for the representation of currencies and funds*.

1 This International Standard is designed to promote the portability of C programs among a variety of data-processing systems. It is intended for use by implementors and programmers. It is accompanied by a Rationale document that explains many of the decisions of the Technical Committee that produced it.

3 Definitions and conventions

In this International Standard, ''shall'' is to be interpreted as a requirement on an implementation or on a program; conversely, ''shall not'' is to be interpreted as a prohibition.

For the purposes of this International Standard, the following definitions apply. Other terms are defined at their first appearance, indicated by *italic* type. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this International Standard are to be interpreted according to ISO 2382.

3.1 alignment: A requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address.

3.2 argument: An expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation. Also known as ''actual argument'' or ''actual parameter.''

3.3 bit: The unit of data storage in the execution environment large enough to hold an object that may have one of two values. It need not be possible to express the address of each individual bit of an object.

3.4 byte: The unit of data storage large enough to hold any member of the basic character set of the execution environment. It shall be possible to express the address of each individual byte of an object uniquely. A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the *low-order* bit; the most significant bit is called the *high-order* bit.

3.5 character: A bit representation that fits in a byte. The representation of each member of the basic character set in both the source and execution environments shall fit in a byte.

3.6 constraints: Syntactic and semantic restrictions by which the exposition of language elements is to be interpreted.

3.7 diagnostic message: A message belonging to an implementation-defined subset of the implementation's message output.

3.8 forward references: References to later subclauses of this International Standard that contain additional information relevant to this subclause.

3.9 implementation: A particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment.

3.10 implementation-defined behavior: Behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation and that each implementation shall document.

3.11 implementation limits: Restrictions imposed upon programs by the implementation.

3.12 locale-specific behavior: Behavior that depends on local conventions of nationality, culture, and language that each implementation shall document.

3.13 multibyte character: A sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment. The extended character set is a superset of the basic character set.

3.14 object: A region of data storage in the execution environment, the contents of which can represent values. Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined. When referenced, an object may be interpreted as having a particular type; see 6.2.2.1.

3.15 parameter: An object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition. Also known as ''formal argument'' or ''formal parameter.''

3.16 undefined behavior: Behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately valued objects, for which this International Standard imposes no requirements. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

If a ''shall'' or ''shall not'' requirement that appears outside of a constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this International Standard by the words ''undefined behavior'' or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe ''behavior that is undefined.''

3.17 unspecified behavior: Behavior, for a correct program construct and correct data, for which this International Standard explicitly imposes no requirements.

Examples

1. An example of unspecified behavior is the order in which the arguments to a function are evaluated.
2. An example of undefined behavior is the behavior on integer overflow.
3. An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.
4. An example of locale-specific behavior is whether the **`islower`** function returns true for characters other than the 26 lowercase English letters.

Forward references: bitwise shift operators (6.3.7), expressions (6.3), function calls (6.3.2.2), the **`islower`** function (7.3.1.6), localization (7.4).

4 Compliance

A *strictly conforming program* shall use only those features of the language and library specified in this International Standard. It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit.

The two forms of *conforming implementation* are hosted and freestanding. A *conforming hosted implementation* shall accept any strictly conforming program. A *conforming freestanding implementation* shall accept any strictly conforming program in which the use of the features specified in the library clause (clause 7) is confined to the contents of the standard headers **`<float.h>`**, **`<limits.h>`**, **`<stdarg.h>`**, and **`<stddef.h>`**. A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.[2]

A *conforming program* is one that is acceptable to a conforming implementation.[3]

---

2 This implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this International Standard.

3 Strictly conforming programs are intended to be maximally portable among conforming implementations. Conforming programs may depend upon nonportable features of a conforming implementation.

An implementation shall be accompanied by a document that defines all implementation-defined characteristics and all extensions.

Forward references: limits **<float.h>** and **<limits.h>** (7.1.5), variable arguments **<stdarg.h>** (7.8), common definitions **<stddef.h>** (7.1.6).

5 Environment

An implementation translates C source files and executes C programs in two data-processing-system environments, which will be called the *translation environment* and the *execution environment* in this International Standard. Their characteristics define and constrain the results of executing conforming C programs constructed according to the syntactic and semantic rules for conforming implementations.

Forward references: In the environment clause (clause 5), only a few of many possible forward references have been noted.

5.1 Conceptual models

5.1.1 Translation environment

5.1.1.1 Program structure

A C program need not all be translated at the same time. The text of the program is kept in units called *source files* in this International Standard. A source file together with all the headers and source files included via the preprocessing directive **#include**, less any source lines skipped by any of the conditional inclusion preprocessing directives, is called a *translation unit*. Previously translated translation units may be preserved individually or in libraries. The separate translation units of a program communicate by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units may be separately translated and then later linked to produce an executable program.

Forward references: conditional inclusion (6.8.1), linkages of identifiers (6.1.2.2), source file inclusion (6.8.2).

5.1.1.2 Translation phases

The precedence among the syntax rules of translation is specified by the following phases.[4]

1. Physical source file characters are mapped to the source character set (introducing new-line characters for end-of-line indicators) if necessary. Trigraph sequences are replaced by corresponding single-character internal representations.

2. Each instance of a new-line character and an immediately preceding backslash character is deleted, splicing physical source lines to form logical source lines. A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character.

3. The source file is decomposed into preprocessing tokens[5] and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or comment. Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.

4. Preprocessing directives are executed and macro invocations are expanded. A **#include** preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively.

---

4 Implementations must behave as if these separate phases occur, even though many are typically folded together in practice.

5 As described in 6.1, the process of dividing a source file's characters into preprocessing tokens is context-dependent. For example, see the handling of **<** within a **#include** preprocessing directive.

5. Each source character set member and escape sequence in character constants and string literals is converted to a member of the execution character set.

6. Adjacent character string literal tokens are concatenated and adjacent wide string literal tokens are concatenated.

7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. The resulting tokens are syntactically and semantically analyzed and translated.

8. All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

Forward references: lexical elements (6.1), preprocessing directives (6.8), trigraph sequences (5.2.1.1).

5.1.1.3 Diagnostics

A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) for every translation unit that contains a violation of any syntax rule or constraint. Diagnostic messages need not be produced in other circumstances.[6]

5.1.2 Execution environments

Two execution environments are defined: *freestanding* and *hosted*. In both cases, *program startup* occurs when a designated C function is called by the execution environment. All objects in static storage shall be *initialized* (set to their initial values) before program startup. The manner and timing of such initialization are otherwise unspecified. *Program termination* returns control to the execution environment.

Forward references: initialization (6.5.7).

5.1.2.1 Freestanding environment

In a freestanding environment (in which C program execution may take place without any benefit of an operating system), the name and type of the function called at program startup are implementation-defined. There are otherwise no reserved external identifiers. Any library facilities available to a freestanding program are implementation-defined.

The effect of program termination in a freestanding environment is implementation-defined.

5.1.2.2 Hosted environment

A hosted environment need not be provided, but shall conform to the following specifications if present.

5.1.2.2.1 Program startup

The function called at program startup is named **`main`**. The implementation declares no prototype for this function. It can be defined with no parameters:

```
int main(void) { /*...*/ }
```

or with two parameters (referred to here as **`argc`** and **`argv`**, though any names may be used, as they are local to the function in which they are declared):

6 The intent is that an implementation should identify the nature of, and where possible localize, each violation. Of course, an implementation is free to produce any number of diagnostics as long as a valid program is still correctly translated. It may also successfully translate an invalid program.

```
int main(int argc, char *argv[]) { /*...*/ }
```

If they are defined, the parameters to the **main** function shall obey the following constraints:

— The value of **argc** shall be nonnegative.

— **argv[argc]** shall be a null pointer.

— If the value of **argc** is greater than zero, the array members **argv[0]** through **argv[argc-1]** inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup. The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment. If the host environment is not capable of supplying strings with letters in both uppercase and lowercase, the implementation shall ensure that the strings are received in lowercase.

— If the value of **argc** is greater than zero, the string pointed to by **argv[0]** represents the *program name*; **argv[0][0]** shall be the null character if the program name is not available from the host environment. If the value of **argc** is greater than one, the strings pointed to by **argv[1]** through **argv[argc-1]** represent the *program parameters*.

— The parameters **argc** and **argv** and the strings pointed to by the **argv** array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.

5.1.2.2.2 Program execution

In a hosted environment, a program may use all the functions, macros, type definitions, and objects described in the library clause (clause 7).

5.1.2.2.3 Program termination

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. If the **main** function executes a return that specifies no value, the termination status returned to the host environment is undefined.

Forward references: definition of terms (7.1.1), the **exit** function (7.10.4.3).

5.1.2.3 Program execution

The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant.

Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment. Evaluation of an expression may produce side effects. At certain specified points in the execution sequence called *sequence points*, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.

In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).

When the processing of the abstract machine is interrupted by receipt of a signal, only the values of objects as of the previous sequence point may be relied on. Objects that may be modified between the previous sequence point and the next sequence point need not have received their correct values yet.

An instance of each object with automatic storage duration is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).