TECHNICAL REPORT

ISO/IEC TR 24731-1

First edition
2007-09-01

Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library

Part 1: **Bounds-checking interfaces**

Technologies de l'information — Langages de programmation, leurs environnements et leurs systèmes d'interface de logiciel — Extensions à la bibliothèque C —

Partie 1: Interfaces des contrôles des bornes

Reference number
ISO/IEC TR 24731-1:2007(E)

Contents

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.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. 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.

In exceptional circumstances, the joint technical committee may propose the publication of a Technical Report of one of the following types:

— type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;

— type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;

— type 3, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 24731-1, which is a Technical Report of type 2, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

ISO/IEC 24731 consists of the following part, under the general title *Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library*:

— *Part 1: Bounds-checking interfaces* [Technical Report]

Introduction

Traditionally, the C library has contained many functions that trust the programmer to provide output character arrays big enough to hold the result being produced. Not only do these functions not check that the arrays are big enough, they frequently lack the information needed to perform such checks. While it is possible to write safe, robust, and error-free code using the existing library, the library tends to promote programming styles that lead to mysterious failures if a result is too big for the provided array.

A common programming style is to declare character arrays large enough to handle most practical cases. However, if these arrays are not large enough to handle the resulting strings, data can be written past the end of the array overwriting other data and program structures. The program never gets any indication that a problem exists, and so never has a chance to recover or to fail gracefully.

Worse, this style of programming has compromised the security of computers and networks. Buffer overflows can often be exploited to run arbitrary code with the permissions of the vulnerable (defective) program.

If the programmer writes runtime checks to verify lengths before calling library functions, then those runtime checks frequently duplicate work done inside the library functions, which discover string lengths as a side effect of doing their job.

This Technical Report provides alternative functions for the C library that promote safer, more secure programming. The functions verify that output buffers are large enough for the intended result and return a failure indicator if they are not. Data is never written past the end of an array. All string results are null terminated.

This Technical Report also addresses another problem that complicates writing robust code: functions that are not re-entrant because they return pointers to static objects owned by the function. Such functions can be troublesome since a previously returned result can change if the function is called again, perhaps by another thread.

Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library —

Part 1: **Bounds-checking interfaces**

1. Scope

This Technical Report specifies a series of extensions of the programming language C, specified by International Standard ISO/IEC 9899:1999. These extensions can be useful in the mitigation of security vulnerabilities in programs, and consist of a new predefined macro, and new functions, macros, and types declared or defined in existing standard headers.

International Standard ISO/IEC 9899:1999 provides important context and specification for this Technical Report. Clauses 3 and 4 of this Technical Report are to be read as if they were merged into Clauses 3 and 4 of ISO/IEC 9899:1999. Clause 5 of this Technical Report is to be read as if it were merged into Subclause 6.10.8 of ISO/IEC 9899:1999. Clause 6 of this Technical Report is to be read as if it were merged into the parallel structure of named Subclauses of Clause 7 of ISO/IEC 9899:1999. Statements made in ISO/IEC 9899:1999, whether about the language or library, apply to this Technical Report unless a corresponding section of this Technical Report states otherwise. In particular, Subclause 7.1.4 ("Use of library functions") of ISO/IEC 9899:1999 applies to this Technical Report.

2. Normative references

The following referenced documents are indispensable for the application 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 9899:1999, *Programming languages — C*

ISO/IEC 9899:1999/Cor.1:2001, *Programming languages — C — Technical Corrigendum 1*

ISO/IEC 9899:1999/Cor.2:2004, *Programming languages — C — Technical Corrigendum 2*

ISO 31−11:1992, *Quantities and units — Part 11: Mathematical signs and symbols for use in the physical sciences and technology*

ISO/IEC 2382−1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms*

3. Terms, definitions, and symbols

For the purposes of this document, the terms and definitions given in ISO/IEC 9899:1999, ISO/IEC 2382−1, and the following apply. Other terms are defined where they appear in *italic* type. Mathematical symbols not defined in this Technical Report are to be interpreted according to ISO 31−11.

3.1
runtime-constraint
requirement on a program when calling a library function

NOTE 1 Despite the similar terms, a runtime-constraint is not a kind of constraint as defined by ISO/IEC 9899:1999, Subclause 3.8, and need not be diagnosed at translation time.

NOTE 2 Implementations verify that the runtime-constraints for a library function are not violated by the program. See Subclause 6.1.4.

4. Conformance

If a ‘‘shall’’ or ‘‘shall not’’ requirement that appears outside of a constraint or runtime-constraint is violated, the behavior is undefined.

5. Predefined macro names

The following macro name is conditionally defined by the implementation:

`__STDC_LIB_EXT1__` The integer constant **`200509L`**, intended to indicate conformance to this Technical Report.[1)]

1) The intention is that this will remain an integer constant of type **`long int`** that is increased with each revision of this Technical Report.

6. Library

6.1 Introduction

6.1.1 Standard headers

The functions, macros, and types declared or defined in Clause 6 and its subclauses are not declared or defined by their respective headers if **`__STDC_WANT_LIB_EXT1__`** is defined as a macro which expands to the integer constant **`0`** at the point in the source file where the appropriate header is included.

The functions, macros, and types declared or defined in Clause 6 and its subclauses are declared and defined by their respective headers if **`__STDC_WANT_LIB_EXT1__`** is defined as a macro which expands to the integer constant **`1`** at the point in the source file where the appropriate header is included.[2)]

It is implementation-defined whether the functions, macros, and types declared or defined in Clause 6 and its subclauses are declared or defined by their respective headers if **`__STDC_WANT_LIB_EXT1__`** is not defined as a macro at the point in the source file where the appropriate header is included.[3)]

Within a preprocessing translation unit, **`__STDC_WANT_LIB_EXT1__`** shall be defined identically for all inclusions of any headers from Clause 6. If **`__STDC_WANT_LIB_EXT1__`** is defined differently for any such inclusion, the implementation shall issue a diagnostic as if a preprocessor error directive was used.

2) Future revisions of this Technical Report may define meanings for other values of **`__STDC_WANT_LIB_EXT1__`**.

3) Subclause 7.1.3 of ISO/IEC 9899:1999 reserves certain names and patterns of names that an implementation may use in headers. All other names are not reserved, and a conforming implementation may not use them. While some of the names defined in Clause 6 and its subclauses are reserved, others are not. If an unreserved name is defined in a header when **`__STDC_WANT_LIB_EXT1__`** is not defined, then the implementation is not conforming.

6.1.2 Reserved identifiers

Each macro name in any of the following subclauses is reserved for use as specified if it is defined by any of its associated headers when included; unless explicitly stated otherwise (see ISO/IEC 9899:1999 Subclause 7.1.4).

All identifiers with external linkage in any of the following subclauses are reserved for use as identifiers with external linkage if any of them are used by the program. None of them are reserved if none of them are used.

Each identifier with file scope listed in any of the following subclauses is reserved for use as a macro name and as an identifier with file scope in the same name space if it is defined by any of its associated headers when included.

6.1.3 Use of errno

An implementation may set **`errno`** for the functions defined in this Technical Report, but is not required to.

6.1.4 Runtime-constraint violations

Most functions in this Technical Report include as part of their specification a list of runtime-constraints. These runtime-constraints are requirements on the program using the library.[4)]

Implementations shall verify that the runtime-constraints for a function are not violated by the program. If a runtime-constraint is violated, the implementation shall call the currently registered runtime-constraint handler (see **`set_constraint_handler_s`** in **`<stdlib.h>`**). Multiple runtime-constraint violations in the same call to a library function result in only one call to the runtime-constraint handler. It is unspecified which one of the multiple runtime-constraint violations cause the handler to be called.

If the runtime-constraints section for a function states an action to be performed when a runtime-constraint violation occurs, the function shall perform the action before calling the runtime-constraint handler. If the runtime-constraints section lists actions that are prohibited when a runtime-constraint violation occurs, then such actions are prohibited to the function both before calling the handler and after the handler returns.

The runtime-constraint handler might not return. If the handler does return, the library function whose runtime-constraint was violated shall return some indication of failure as given by the returns section in the function's specification.

4) Although runtime-constraints replace many cases of undefined behavior from International Standard ISO/IEC 9899:1999, undefined behavior still exists in this Technical Report. Implementations are free to detect any case of undefined behavior and treat it as a runtime-constraint violation by calling the runtime-constraint handler. This license comes directly from the definition of undefined behavior.

6.2 Errors `<errno.h>`

The header **`<errno.h>`** defines a type.

The type is

```
    errno_t
```

which is type **`int`**.[5)]

5) As a matter of programming style, **`errno_t`** may be used as the type of something that deals only with the values that might be found in **`errno`**. For example, a function which returns the value of **`errno`** might be declared as having the return type **`errno_t`**.

6.3 Common definitions `<stddef.h>`

The header **`<stddef.h>`** defines a type.

The type is

```
    rsize_t
```

which is the type **`size_t`**.[6)]

6) See the description of the **`RSIZE_MAX`** macro in **`<stdint.h>`**.

6.4 Integer types `<stdint.h>`

The header **`<stdint.h>`** defines a macro.

The macro is

`RSIZE_MAX`

which expands to a value[7)] of type **`size_t`**. Functions that have parameters of type **`rsize_t`** consider it a runtime-constraint violation if the values of those parameters are greater than **`RSIZE_MAX`**.

Recommended practice

Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like **`size_t`**. Also, some implementations do not support objects as large as the maximum value that can be represented by type **`size_t`**.

For those reasons, it is sometimes beneficial to restrict the range of object sizes to detect programming errors. For implementations targeting machines with large address spaces, it is recommended that **`RSIZE_MAX`** be defined as the smaller of the size of the largest object supported or **`(SIZE_MAX >> 1)`**, even if this limit is smaller than the size of some legitimate, but very large, objects. Implementations targeting machines with small address spaces may wish to define **`RSIZE_MAX`** as **`SIZE_MAX`**, which means that there is no object size that is considered a runtime-constraint violation.

7) The macro **`RSIZE_MAX`** need not expand to a constant expression.

6.5 Input/output `<stdio.h>`

The header **`<stdio.h>`** defines several macros and two types.

The macros are

    L_tmpnam_s

which expands to an integer constant expression that is the size needed for an array of **`char`** large enough to hold a temporary file name string generated by the **`tmpnam_s`** function;

    TMP_MAX_S

which expands to an integer constant expression that is the maximum number of unique file names that can be generated by the **`tmpnam_s`** function.

The types are

    errno_t

which is type **`int`**; and

    rsize_t

which is the type **`size_t`**.

6.5.1 Operations on files

6.5.1.1 The `tmpfile_s` function

Synopsis

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
errno_t tmpfile_s(FILE * restrict * restrict streamptr);
```

Runtime-constraints

`streamptr` shall not be a null pointer.

If there is a runtime-constraint violation, **`tmpfile_s`** does not attempt to create a file.

Description

The **`tmpfile_s`** function creates a temporary binary file that is different from any other existing file and that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with **`"wb+"`** mode with the meaning that mode has in the **`fopen_s`** function (including the mode's effect on exclusive access and file permissions).

If the file was created successfully, then the pointer to **FILE** pointed to by **streamptr** will be set to the pointer to the object controlling the opened file. Otherwise, the pointer to **FILE** pointed to by **streamptr** will be set to a null pointer.

Recommended practice

It should be possible to open at least **TMP_MAX_S** temporary files during the lifetime of the program (this limit may be shared with **tmpnam_s**) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files (**FOPEN_MAX**).

Returns

The **tmpfile_s** function returns zero if it created the file. If it did not create the file or there was a runtime-constraint violation, **tmpfile_s** returns a non-zero value.

6.5.1.2 The tmpnam_s function

Synopsis

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
errno_t tmpnam_s(char *s, rsize_t maxsize);
```

Runtime-constraints

s shall not be a null pointer. **maxsize** shall be less than or equal to **RSIZE_MAX**. **maxsize** shall be greater than the length of the generated file name string.

Description

The **tmpnam_s** function generates a string that is a valid file name and that is not the same as the name of an existing file.[8] The function is potentially capable of generating **TMP_MAX_S** different strings, but any or all of them may already be in use by existing files and thus not be suitable return values. The lengths of these strings shall be less than the value of the **L_tmpnam_s** macro.

The **tmpnam_s** function generates a different string each time it is called.

8) Files created using strings generated by the **tmpnam_s** function are temporary only in the sense that their names should not collide with those generated by conventional naming rules for the implementation. It is still necessary to use the **remove** function to remove such files when their use is ended, and before program termination. Implementations should take care in choosing the patterns used for names returned by **tmpnam_s**. For example, making a thread id part of the names avoids the race condition and possible conflict when multiple programs run simultaneously by the same user generate the same temporary file names.