ISO/IEC JTC1 SC22 WG14 N1088

Date: 2004-11-30

Reference number of document: **ISO/IEC WDTR 24731**

Committee identification: ISO/IEC JTC1 SC22 WG14

SC22 Secretariat: ANSI

Information Technology —

Programming languages, their environments and system software interfaces —

Specification for Secure C Library Functions —

Warning

This document is an ISO/IEC draft Technical Report. It is not an ISO/IEC International Technical Report. It is distributed for review and comment. It is subject to change without notice and shall not be referred to as an International Technical Report or International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: Technical Report Type 2
Document subtype: n/a
Document stage: (2) Working Draft
Document language: E

Contents

Foreword

1 ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are member 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.

2 Technical Reports are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft Technical Reports adopted by the joint technical committee are circulated to national bodies for voting. Publication as a Technical Report requires approval by at least 75% of the member bodies casting a vote.

3 The main task of technical committees is to prepare International Standards, but in exceptional circumstances a 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 a 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).

4 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.

5 ISO/IEC TR 24731, 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*.

Introduction

1 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.

2 Perhaps the most common programming style is to declare character arrays large enough to handle most practical cases. However, if the program encounters strings too large for it to process, data is written past the end of arrays overwriting other variables in the program. The program never gets any indication that a problem exists, and so never has a chance to recover or to fail gracefully.

3 Worse, this style of programming has compromised the security of computers and networks. Daemons are given carefully prepared data that overflows buffers and tricks the daemons into granting access that should be denied.

4 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.

5 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.

6 This technical report also addresses another problem that complicates writing robust code: functions that are not reentrant 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. *

1. Scope

1 This Technical Report specifies a series of extensions of the programming language C, specified by International Standard ISO/IEC 9899:1999.

2 International Standard ISO/IEC 9899:1999 provides important context and specification for this Technical Report. Clause 4 of this Technical Report should be read as if it was merged into Subclause 6.10.8 of ISO/IEC 9899:1999. Clause 5 of this Technical Report should be read as if it was merged into the parallel structure of named Subclauses of Clause 7 of ISO/IEC 9899:1999.

2. Normative references

1 The following normative documents contain provisions which, through reference in this text, constitute provisions of this Technical Report. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this Technical Report are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

2 ISO/IEC 9899:1999, *Information technology — Programming languages, their environments and system software interfaces — Programming Language C*.

3 ISO/IEC 9899:1999/Cor 1:2001, *Information technology — Programming languages, their environments and system software interfaces — Programming Language C — Technical Corrigendum 1* .

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

5 ISO/IEC 646, *Information technology — ISO 7-bit coded character set for information interchange*.

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

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

8 ISO 8601, *Data elements and interchange formats — Information interchange — Representation of dates and times*.

9 ISO/IEC 10646 (all parts), *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*.

10 IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems* (previously designated IEC 559:1989).

3. Terms, definitions, and symbols

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

3.1

1 **diagnosed undefined behavior**
behavior, upon use of a nonportable or erroneous program construct or of erroneous data, that an implementation shall diagnose by, in effect, calling an implementation-defined function[1)] (including for this purpose, the **`abort`** macro). The called function shall not have any undefined behavior.[2)]

2 This Technical Report explicitly states the conditions that constitute diagnosed undefined behavior.[3)] Typically, these conditions are invalid values as arguments to a library function upon entry to that function. This technical report then states how the function should behave in the presence of the diagnosed undefined behavior.

3 Should the implementation-defined called function return, the function containing the diagnosed undefined behavior will then behave as explicitly specified for the case of diagnosed undefined behavior.[4)]

4 If an invocation of a library function contains multiple instances of diagnosed undefined behavior, only the first such instance encountered need be diagnosed by the implementation, although the implementation is permitted to diagnose others. An implementation shall prevent undefined behavior by diagnosing any diagnosed undefined behavior leading to that undefined behavior.

---

1) For example, the implementation-defined called function might do nothing, print a message, exit the program, call a user-written handler, or raise a signal.

2) If the implementation-defined called function invokes a user-written function (either by calling it or as a signal handler), there is always the possibility that the user-written function might contain undefined behavior. That is permitted since it is not undefined behavior in the called function itself.

3) Thus, an implementation is only required to diagnose "diagnosed undefined behavior." Of course, an implementation could choose to treat "plain" undefined behavior as diagnosed undefined behavior since there is no requirements on what an implementation does with "plain" undefined behavior.

4) Typically, the library function will return some sort of error indicator immediately after detecting the diagnosed undefined behavior.

5 EXAMPLE The call **strcpy_s(NULL, 10, NULL)** contains two instances of diagnosed undefined behavior. At least one of them will be diagnosed by the implementation. The call will not indirect through either null pointer to load or store a value. If the call to **strcpy_s** returns, it returns a non-zero value to indicate failure.

4. Predefined macro names

1 The following macro name is conditionally defined by the implementation:

__STDC_SECURE_LIB__ The integer constant **200411L**, intended to indicate conformance to this technical report.[5)]

5) The intention is that this will remain an integer constant of type **long int** that is increased with each revision of this technical report.

5. Library

5.1 Introduction

5.1.1 Standard headers

1 The functions, macros, and types defined in Clause 5 and its subclauses are not defined by their respective headers if **__STDC_WANT_SECURE_LIB__** 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.

2 The functions, macros, and types defined in Clause 5 and its subclauses are defined by their respective headers if **__STDC_WANT_SECURE_LIB__** 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.[6)]

3 It is implementation-defined whether the functions, macros, and types defined in Clause 5 and its subclauses are defined by their respective headers if **__STDC_WANT_SECURE_LIB__** is not defined as a macro at the point in the source file where the appropriate header is included.[7)]

4 Within a preprocessing translation unit, **__STDC_WANT_SECURE_LIB__** shall be defined identically for all inclusions of any headers from Clause 5. If **__STDC_WANT_SECURE_LIB__** is defined differently for any such inclusion, the implementation shall issue a diagnostic as if a preprocessor error directive was used.

---

6) Future revisions of this technical report may define meanings for other values of **__STDC_WANT_SECURE_LIB__**.

7) 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 5 and its subclauses are reserved, others are not. If an unreserved name is defined in a header when **__STDC_WANT_SECURE_LIB__** is not defined, then the implementation is not conforming.

5.1.2 Reserved identifiers

1 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).

2 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.

3 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.

5.1.3 Use of errno

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

5.2 Errors `<errno.h>`

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

2 The type is

`errno_t`

which is type **`int`**.[8)]

8) 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`**.

5.3 Integer types <stdint.h>

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

2 The macro is

RSIZE_MAX

which expands to a value[9] of type **size_t**. Functions that have parameters of type **rsize_t** consider it diagnosable undefined behavior if the values of those parameters are greater than **RSIZE_MAX**.

Recommended practice

3 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**.

4 For those reasons, it is sometimes beneficial to restrict the range of object sizes in order to detect bugs. 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 no object sizes are considered diagnosable undefined behavior.

---

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

5.4 Input/output <stdio.h>

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

2 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.

3 The types are

errno_t

which is type **int**; and

rsize_t

which is the type **size_t**.

5.4.1 Operations on files

5.4.1.1 The tmpfile_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdio.h>
errno_t tmpfile_s(FILE **stream);
```

Description

2 If **stream** is null, then it is diagnosed undefined behavior, and **tmpfile_s** does not attempt to create a file or to indirect through **stream**.

3 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.

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

5 The **tmpfile_s** function returns zero if it created the file. If it did not create the file or there was diagnosed undefined behavior, **tmpfile_s** returns a non-zero value.

5.4.1.2 The tmpnam_s function

Synopsis

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

Description

2 If **s** is a null pointer or **maxsize**> **RSIZE_MAX** or **maxsize** is not greater than the length of the generated file name string then it is diagnosed undefined behavior.

3 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.[10)] 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.

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

5 The implementation shall behave as if no library function except **tmpnam** calls the **tmpnam_s** function.[11)]

---

10) 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.

11) An implementation may have **tmpnam** call **tmpnam_s** (perhaps so there is only one naming convention for temporary files), but this is not required.

Returns

6 If no suitable string can be generated, or if there is diagnosed undefined behavior, the **tmpnam_s** function writes a null character to **s[0]** (only if **s** is not null and **maxsize** is greater than zero) and returns a non-zero value.

7 Otherwise, the **tmpnam_s** function writes the string in the array pointed to by **s** and returns zero.

Environmental limits

8 The value of the macro **TMP_MAX_S** shall be at least 25.

5.4.2 File access functions

5.4.2.1 The fopen_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdio.h>
errno_t fopen_s(FILE * restrict * restrict stream,
     const char * restrict filename,
     const char * restrict mode);
```

Description

2 There is diagnosed undefined behavior if **stream**, **filename**, or **mode** is a null pointer. In which case, **fopen_s** does not attempt to open a file or to indirect through **stream**.

3 The **fopen_s** function opens the file whose name is the string pointed to by **filename**, and associates a stream with it. The **mode** argument is used just as in the **fopen** function.

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

Returns

5 The **fopen_s** function returns zero if it opened the file. If it did not open the file or there was diagnosed undefined behavior, **fopen_s** returns a non-zero value.

5.4.2.2 The `freopen_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdio.h>
errno_t freopen_s(FILE * restrict * restrict newstream,
     const char * restrict filename,
     const char * restrict mode,
     FILE * restrict stream);
```

Description

2 There is diagnosed undefined behavior if **`newstream`**, **`mode`**, or **`stream`** is a null pointer. If there is diagnosed undefined behavior, **`freopen_s`** does not attempt to open a file or to indirect through **`newstream`**.

3 The **`freopen_s`** function opens the file whose name is the string pointed to by **`filename`** and associates the stream pointed to by **`stream`** with it. The **`mode`** argument is used just as in the **`fopen`** function.

4 If **`filename`** is a null pointer, the **`freopen_s`** function attempts to change the mode of the stream to that specified by **`mode`**, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

5 The **`freopen_s`** function first attempts to close any file that is associated with **`stream`**. Failure to close the file is ignored. The error and end-of-file indicators for the stream are cleared.

6 If the file was opened successfully, then the pointer to **`FILE`** pointed to by **`newstream`** will be set to the value of **`stream`**. Otherwise, the pointer to **`FILE`** pointed to by **`newstream`** will be set to a null pointer.

Returns

7 The **`freopen_s`** function returns zero if it opened the file. If it did not open the file or there was diagnosed undefined behavior, **`freopen_s`** returns a non-zero value.

5.4.3 Formatted input/output functions

5.4.3.1 The `fscanf_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdio.h>
int fscanf_s(FILE * restrict stream,
     const char * restrict format, ...);
```

Description

2 If either **`stream`** or **`format`** is a null pointer, there is diagnosed undefined behavior, and the **`fscanf_s`** function does not attempt to perform input.

3 The **`fscanf_s`** function is equivalent to **`fscanf`** except that the **`c`**, **`s`**, and **`[`** conversion specifiers apply to a pair of arguments (unless assignment suppression is indicated by a **`*`**). The first of these arguments is the same as for **`fscanf`**. That argument is immediately followed in the argument list by the second argument, which has type **`rsize_t`** and gives the number of elements in the array pointed to by the first argument of the pair. If the first argument points to a scalar object, it is considered to be an array of one element.[12)]

4 A matching failure occurs if the number of elements in a receiving object is insufficient to hold the converted input (including any trailing null character).

Returns

5 The **`fscanf_s`** function returns the value of the macro **`EOF`** if an input failure occurs before any conversion or if there is diagnosed undefined behavior. Otherwise, the **`fscanf_s`** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

---

12) If the format is known at translation time, an implementation may issue a diagnostic for any argument used to store the result from a **`c`**, **`s`**, or **`[`** conversion specifier if that argument is not followed by an argument of a type compatible with **`rsize_t`**. A limited amount of checking may be done if even if the format is not known at translation time. For example, an implementation may issue a diagnostic for each argument after **`format`** that has of type pointer to one of **`char`**, **`signed char`**, **`unsigned char`**, or **`void`** that is not followed by an argument of a type compatible with **`rsize_t`**. The diagnostic could warn that unless the pointer is being used with a conversion specifier using the **`hh`** length modifier, a length argument must follow the pointer argument. Another useful diagnostic could flag any non-pointer argument following **`format`** that did not have a type compatible with **`rsize_t`**.

6 EXAMPLE 1 The call:

```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdio.h>
/* ... */
int n, i; float x; char name[50];
n = fscanf_s(stdin, "%d%f%s", &i, &x, name, (rsize_t) 50);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to **n** the value 3, to **i** the value 25, to **x** the value 5.432, and to **name** the sequence **thompson\0**.

7 EXAMPLE 2 The call:

```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdio.h>
/* ... */
int n; char s[5];
n = fscanf_s(stdin, "%s", s, sizeof s);
```

with the input line:

```
hello
```

will assign to **n** the value 0 since a matching failure occurred because the sequence **hello\0** requires an array of six characters to store it. No assignment to **s** occurs.

5.4.3.2 The scanf_s function

Synopsis

1

```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdio.h>
int scanf_s(const char * restrict format, ...);
```

Description

2 If **format** is a null pointer, there is diagnosed undefined behavior, and the **scanf_s** function does not attempt to perform input.

3 The **scanf_s** function is equivalent to **fscanf_s** with the argument **stdin** interposed before the arguments to **scanf_s**.

Returns

4 The **scanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is diagnosed undefined behavior. Otherwise, the **scanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

5.4.3.3 The `sscanf_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdio.h>
int sscanf_s(const char * restrict s,
     const char * restrict format, ...);
```

Description

2 If either **s** or **format** is a null pointer, there is diagnosed undefined behavior, and the **sscanf_s** function does not attempt to perform input.

3 The **sscanf_s** function is equivalent to **fscanf_s**, except that input is obtained from a string (specified by the argument **s**) rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the **fscanf_s** function. If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

4 The **sscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is diagnosed undefined behavior. Otherwise, the **sscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

5.4.3.4 The `vfscanf_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdarg.h>
#include <stdio.h>
int vfscanf_s(FILE * restrict stream,
     const char * restrict format,
     va_list arg);
```

Description

2 If either **stream** or **format** is a null pointer, there is diagnosed undefined behavior, and the **vfscanf_s** function does not attempt to perform input.

3 The **vfscanf_s** function is equivalent to **fscanf_s**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfscanf_s** function does not invoke the **va_end** macro.[13)]

Returns

4 The **vfscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is diagnosed undefined behavior. Otherwise, the **vfscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

5.4.3.5 The vscanf_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdarg.h>
#include <stdio.h>
int vscanf_s(const char * restrict format,
     va_list arg);
```

Description

2 If **format** is a null pointer, there is diagnosed undefined behavior, and the **vscanf_s** function does not attempt to perform input.

3 The **vscanf_s** function is equivalent to **scanf_s**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vscanf_s** function does not invoke the **va_end** macro.[13)]

Returns

4 The **vscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is diagnosed undefined behavior. Otherwise, the **vscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

5.4.3.6 The vsscanf_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdarg.h>
#include <stdio.h>
int vsscanf_s(const char * restrict s,
     const char * restrict format,
     va_list arg);
```

13) As the functions **vfscanf_s**, **vscanf_s**, and **vsscanf_s** invoke the **va_arg** macro, the value of **arg** after the return is indeterminate.

Description

2 If either **s** or **format** is a null pointer, there is diagnosed undefined behavior, and the **fscanf_s** function does not attempt to perform input.

3 The **vsscanf_s** function is equivalent to **sscanf_s**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsscanf_s** function does not invoke the **va_end** macro.[13)]

Returns

4 The **vsscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is diagnosed undefined behavior. Otherwise, the **vscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

5.4.4 Character input/output functions

5.4.4.1 The gets_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdio.h>
char *gets_s(char *s, rsize_t n);
```

Description

2 If **n** is equal to zero or **s** is a null pointer or **n > RSIZE_MAX**, then there is diagnosed undefined behavior, and no input is performed and the array pointed to by **s** (if any) is not modified.

3 Otherwise, the **gets_s** function reads at most one less than the number of characters specified by **n** from the stream pointed to by **stdin**, into the array pointed to by **s**. No additional characters are read after a new-line character (which is discarded) or after end-of-file. Although a new-line character counts towards number of characters read, it is not stored in the array. A null character is written immediately after the last character read into the array.

4 If end-of-file is encountered and no characters have been read into the array, or if a read error occurs during the operation, then **s[0]** is set to the null character.

Returns

5 The **gets_s** function returns **s** if successful. If there was diagnosed undefined behavior, or if end-of-file is encountered and no characters have been read into the array, or if a read error occurs during the operation, then a null pointer is returned.

5.5 General utilities <stdlib.h>

1 The header **<stdlib.h>** defines two types.

2 The types are

errno_t

which is type **int**; and

rsize_t

which is the type **size_t**.

5.5.1 Communication with the environment

5.5.1.1 The getenv_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdlib.h>
errno_t getenv_s(size_t * restrict needed,
          char * restrict value, rsize_t maxsize,
          const char * restrict name);
```

Description

2 There is diagnosed undefined behavior if any of the following conditions are true:

— **name == NULL**

— **maxsize >** RSIZE_MAX

— **maxsize > 0 && value == NULL**

If there is diagnosed undefined behavior, the integer pointed to by **needed** is set to 0 (if **needed** is not null), and the function returns.

3 The **getenv_s** function searches an *environment list*, provided by the host environment, for a string that matches the string pointed to by **name**.

4 If that name is found then **getenv_s** performs the following actions. If **needed** is not a null pointer, one plus the length of the string associated with the matched list member is stored in the integer pointed to by **needed**. If the length of the associated string is less than **maxsize**, then the associated string is copied to the array pointed to by **value**.

5 If that name is not found then **getenv_s** performs the following actions. If **needed** is not a null pointer, zero is stored in the integer pointed to by **needed**. If **maxsize** is greater than zero, then **value[0]** is set to the null character.

6 The set of environment names and the method for altering the environment list are implementation-defined.

Returns

7 The **getenv_s** function returns zero if the specified **name** is found and the associated string was successfully stored in **value**. Otherwise, a non-zero value is returned.

5.5.2 Searching and sorting utilities

1 These utilities make use of a comparison function to search or sort arrays of unspecified type. Where an argument declared as **size_t nmemb** specifies the length of the array for a function, **nmemb** can have the value zero on a call to that function; the comparison function is not called, a search finds no matching element, sorting performs no rearrangement, and the pointer to the array may be null.

2 The implementation shall ensure that the second argument of the comparison function (when called from **bsearch_s**), or both arguments (when called from **qsort_s**), are pointers to elements of the array.[14)] The first argument when called from **bsearch_s** shall equal **key**.

3 The comparison function shall not alter the contents of either the array or search key. The implementation may reorder elements of the array between calls to the comparison function, but shall not otherwise alter the contents of any individual element.

4 When the same objects (consisting of **size** bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, for **qsort_s** they shall define a total ordering on the array, and for **bsearch_s** the same object shall always compare the same way with the key.

5 A sequence point occurs immediately before and immediately after each call to the comparison function, and also between any call to the comparison function and any movement of the objects passed as arguments to that call.

---

14) That is, if the value passed is **p**, then the following expressions are always valid and nonzero:

```
((char *)p - (char *)base) % size == 0
(char *)p >= (char *)base
(char *)p < (char *)base + nmemb * size
```

5.5.2.1 The `bsearch_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdlib.h>
void *bsearch_s(const void *key, const void *base,
     rsize_t nmemb, rsize_t size,
     int (*compar)(const void *k, const void *y,
                   void *context),
     void *context);
```

Description

2 There is diagnosed undefined behavior if any of the following conditions are true:

— **`nmemb > RSIZE_MAX`**

— **`size > RSIZE_MAX`**

— **`nmemb!=0 && (key==NULL || base==NULL || compar==NULL || compar==NULL)`**

In which case, the **`bsearch_s`** function does not search the array.

3 The **`bsearch_s`** function searches an array of **`nmemb`** objects, the initial element of which is pointed to by **`base`**, for an element that matches the object pointed to by **`key`**. The size of each element of the array is specified by **`size`**.

4 The comparison function pointed to by **`compar`** is called with three arguments. The first two point to the **`key`** object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the **`key`** object is considered, respectively, to be less than, to match, or to be greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the **`key`** object, in that order.[15)] The third argument to the comparison function is the **`context`** argument passed to **`bsearch_s`**. The sole use of **`context`** by **`bsearch_s`** is to pass it to the comparison function.[16)]

Returns

5 The **`bsearch_s`** function returns a pointer to a matching element of the array, or a null pointer if no match is found or there is diagnosable undefined behavior. If two elements

---

15) In practice, the entire array has been sorted according to the comparison function.

16) The **`context`** argument is for the use of the comparison function in performing its duties. For example, it might specify a collating sequence used by the comparison function.

compare as equal, which element is matched is unspecified.

5.5.2.2 The `qsort_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdlib.h>
void qsort_s(void *base, rsize_t nmemb, rsize_t size,
     int (*compar)(const void *x, const void *y,
                   void *context),
     void *context);
```

Description

2 There is diagnosed undefined behavior if any of the following conditions are true:

— **`nmemb > RSIZE_MAX`**

— **`size > RSIZE_MAX`**

— **`nmemb != 0 && (base == NULL || compar == NULL)`**

In which case, the **`qsort_s`** function does not sort the array.

3 The **`qsort_s`** function sorts an array of **`nmemb`** objects, the initial element of which is pointed to by **`base`**. The size of each object is specified by **`size`**.

4 The contents of the array are sorted into ascending order according to a comparison function pointed to by **`compar`**, which is called with three arguments. The first two point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. The third argument to the comparison function is the **`context`** argument passed to **`qsort_s`**. The sole use of **`context`** by **`qsort_s`** is to pass it to the comparison function.[16)]

5 If two elements compare as equal, their relative order in the resulting sorted array is unspecified.

Returns

6 The **`qsort_s`** function returns no value.

5.5.3 Multibyte/wide character conversion functions

1 The behavior of the multibyte character functions is affected by the **LC_CTYPE** category of the current locale. For a state-dependent encoding, each function is placed into its initial conversion state by a call for which its character pointer argument, **s**, is a null pointer. Subsequent calls with **s** as other than a null pointer cause the internal conversion state of the function to be altered as necessary. A call with **s** as a null pointer causes these functions to set the int pointed to by their **status** argument to a nonzero value if encodings have state dependency, and zero otherwise.[17] Changing the **LC_CTYPE** category causes the conversion state of these functions to be indeterminate.

5.5.3.1 The wctomb_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdlib.h>
errno_t wctomb_s(int * restrict status,
     char * restrict s,
     rsize_t smax,
     wchar_t wc);
```

Description

2 Let *n* denote the number of bytes needed to represent the multibyte character corresponding to the wide character given by **wc** (including any shift sequences).

3 There is diagnosed undefined behavior if any of the following conditions are true:

— **(s == NULL && smax != 0)**

— **(smax > RSIZE_MAX**)

— **(s != NULL &&** *n* **>= smax)**

If there is diagnosed undefined behavior, **wctomb_s** does not modify the int pointed to by **status**, and if **s** is not null, no more than **smax** elements in the array pointed to by **s** will be accessed.

4 The **wctomb_s** function determines *n* and stores the multibyte character representation of **wc** in the array whose first element is pointed to by **s** (if **s** is not a null pointer). The number of characters stored never exceeds **MB_CUR_MAX** or **smax**. If **wc** is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state, and the function is left in the initial conversion state.

17) If the locale employs special bytes to change the shift state, these bytes do not produce separate wide character codes, but are grouped with an adjacent multibyte character.

5 The implementation shall behave as if no library function calls the **wctomb_s** function.

6 If **s** is a null pointer, the **wctomb_s** function stores into the int pointed to by **status** a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings.

7 If **s** is not a null pointer, the **wctomb_s** function stores into the int pointed to by **status** either *n* or −1 if **wc**, respectively, does or does not correspond to a valid multibyte character.

8 In no case will the int pointed to by **status** be set to a value greater than the **MB_CUR_MAX** macro.

Returns

9 The **wctomb_s** function returns zero if successful, and a non-zero value if there was diagnosed undefined behavior or **wc** did not correspond to a valid multibyte character.

5.6 String handling <string.h>

1 The header **<string.h>** defines two types.

2 The types are

errno_t

which is type **int**; and

rsize_t

which is the type **size_t**.

5.6.1 Copying functions

5.6.1.1 The memcpy_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <string.h>
errno_t memcpy_s(void * restrict s1, rsize_t s1max,
     const void * restrict s2, rsize_t n);
```

Description

2 There is diagnosed undefined behavior if any of the following conditions are true:

— **s1 == NULL || s2 == NULL**

— **s1max > RSIZE_MAX || n > RSIZE_MAX**

— **n > s1max**

If there is diagnosed undefined behavior, the **memcpy_s** function stores zeros in the first **s1max** characters of the object pointed to by **s1** if **s1 != NULL && s1max <= RSIZE_MAX**

3 Otherwise, the **memcpy_s** function copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. *

4 If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

5 The **memcpy_s** function returns zero if there was no diagnosed undefined behavior. Otherwise, a non-zero value is returned.

5.6.1.2 The `memmove_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <string.h>
errno_t memmove_s(void *s1, rsize_t s1max,
     const void *s2, rsize_t n);
```

Description

2 There is diagnosed undefined behavior if any of the following conditions are true:

— **`s1 == NULL || s2 == NULL`**

— **`s1max > RSIZE_MAX || n > RSIZE_MAX`**

— **`n > s1max`**

If there is diagnosed undefined behavior, the **`memmove_s`** function stores zeros in the first **`s1max`** characters of the object pointed to by **`s1`** if **`s1 != NULL && s1max <= RSIZE_MAX`**

3 Otherwise, the **`memmove_s`** function copies **`n`** characters from the object pointed to by **`s2`** into the object pointed to by **`s1`**. This copying takes place as if the **`n`** characters from the object pointed to by **`s2`** are first copied into a temporary array of **`n`** characters that does not overlap the objects pointed to by **`s1`** or **`s2`**, and then the **`n`** characters from the temporary array are copied into the object pointed to by **`s1`**.

Returns

4 The **`memmove_s`** function returns zero if there was no diagnosed undefined behavior. Otherwise, a non-zero value is returned.

5.6.1.3 The `strcpy_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <string.h>
errno_t strcpy_s(char * restrict s1,
     rsize_t s1max,
     const char * restrict s2);
```

Description

2 There is diagnosed undefined behavior if any of the following conditions are true:

— **`s1 == NULL || s2 == NULL`**

— **s1max > RSIZE_MAX**

— **s1max == 0**

— **s1max <= strnlen_s(s2, s1max)**

If there is diagnosed undefined behavior, then **s1[0]** is set to the null character if **s1 != NULL && s1max > 0 && s1max <= RSIZE_MAX**.

3 Otherwise, the characters pointed to by **s2** up to and including the null character are copied to the array pointed to by **s1**.

4 All elements following the terminating null character (if any) written by **strcpy_s** in the array of **s1max** characters pointed to by **s1** take unspecified values when **strcpy_s** returns.[18)]

5 If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

6 The **strcpy_s** function returns zero[19)] if there was no diagnosed undefined behavior. Otherwise, a non-zero value is returned.

5.6.1.4 The strncpy_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <string.h>
errno_t strncpy_s(char * restrict s1,
     rsize_t s1max,
     const char * restrict s2,
     rsize_t n);
```

Description

2 There is diagnosed undefined behavior if any of the following conditions are true:

— **s1 == NULL || s2 == NULL**

18) This allows an implementation to copy characters from **s2** to **s1** while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of **s1** before discovering that the first element should be set to the null character.

19) A zero return value implies that all of the requested characters from the string pointed to by **s2** fit within the array pointed to by **s1** and that the result in **s1** is null terminated.

— **s1max > RSIZE_MAX || n > RSIZE_MAX**,

— **s1max == 0**

— **n >= s1max && s1max <= strnlen_s(s2, s1max)**

If there is diagnosed undefined behavior, then **s1[0]** is set to the null character if **s1 != NULL && s1max > 0 && s1max <= RSIZE_MAX**.

3 Otherwise, the **strncpy_s** function copies not more than **n** successive characters (characters that follow a null character are not copied) from the array pointed to by **s2** to the array pointed to by **s1**. If no null character was copied from **s2**, then **s1[n]** is set to a null character.

4 All elements following the terminating null character (if any) written by **strncpy_s** in the array of **s1max** characters pointed to by **s1** take unspecified values when **strncpy_s** returns.[20)]

5 If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

6 The **strncpy_s** function returns zero[21)] if there was no diagnosed undefined behavior. Otherwise, a non-zero value is returned.

7 EXAMPLE 1 The **strncpy_s** function can be used to copy a string without the danger that the result will not be null terminated or that characters will be written past the end of the destination array.

```
#define __STDC_WANT_SECURE_LIB__ 1
#include <string.h>
/* … */
char src1[100] = "hello";
char src2[7] = {'g', 'o', 'o', 'd', 'b', 'y', 'e'};
char dst1[6], dst2[5], dst3[5];
int r1, r2, r3;
r1 = strncpy_s(dst1, 6, src1, 100);
r2 = strncpy_s(dst2, 5, src2, 7);
r3 = strncpy_s(dst3, 5, src2, 4);
```

The first call will assign to **r1** the value zero and to **dst1** the sequence **hello\0**.
The second call will assign to **r2** the value **ERANGE** and to **dst2** the sequence **\0**.
The third call will assign to **r3** the value zero and to **dst3** the sequence **good\0**.

---

20) This allows an implementation to copy characters from **s2** to **s1** while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of **s1** before discovering that the first element should be set to the null character.

21) A zero return value implies that all of the requested characters from the string pointed to by **s2** fit within the array pointed to by **s1** and that the result in **s1** is null terminated.

5.6.2 Concatenation functions

5.6.2.1 The `strcat_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <string.h>
errno_t strcat_s(char * restrict s1,
     rsize_t s1max,
     const char * restrict s2);
```

Description

2 Let *m* denote the value **s1max - strnlen_s(s1, s1max)** upon entry to **strcat_s**.

3 There is diagnosed undefined behavior if any of the following conditions are true:

— **s1 == NULL || s2 == NULL**

— **s1max > RSIZE_MAX**

— **s1max == 0**

— *m* **== 0**[22)]

— *m* **<= strnlen(s2,** *m***)**,

If there is diagnosed undefined behavior, then **s1[0]** is set to the null character if **s1 != NULL && s1max > 0 && s1max <= RSIZE_MAX**.

4 Otherwise, the characters pointed to by **s2** up to and including the null character are appended to the end of the string pointed to by **s1**. The initial character from **s2** overwrites the null character at the end of **s1**.

5 All elements following the terminating null character (if any) written by **strcat_s** in the array of **s1max** characters pointed to by **s1** take unspecified values when **strcat_s** returns.[23)] *

6 If copying takes place between objects that overlap, the objects take on unspecified values.

---

22) This means that **s1** was not null terminated upon entry to **strcat_s**.

23) This allows an implementation to append characters from **s2** to **s1** while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of **s1** before discovering that the first element should be set to the null character.

Returns

7 The **strcat_s** function returns zero[24] if there was no diagnosed undefined behavior. Otherwise, a non-zero value is returned.

5.6.2.2 The strncat_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <string.h>
errno_t strncat_s(char * restrict s1,
     rsize_t s1max,
     const char * restrict s2,
     rsize_t n);
```

Description

2 Let *m* denote the value **s1max - strnlen_s(s1, s1max)** upon entry to **strncat_s.**

3 There is diagnosed undefined behavior if any of the following conditions are true:

— **s1 == NULL || s2 == NULL**

— **s1max > RSIZE_MAX || n > RSIZE_MAX**,

— **s1max == 0**

— *m* **== 0**[25]

— **n >=** *m* **&&** *m* **<= strnlen_s(s2,** *m***)**,

If there is diagnosed undefined behavior, then **s1[0]** is set to the null character if **s1 != NULL && s1max > 0 && s1max <= RSIZE_MAX**.

4 Otherwise, the **strncat_s** function appends not more than **n** successive characters (characters that follow a null character are not copied) from the array pointed to by **s2** to the end of the string pointed to by **s1**. The initial character from **s2** overwrites the null character at the end of **s1**. If no null character was copied from **s2**, then **s1[s1max-m+n]** is set to a null character.

5 All elements following the terminating null character (if any) written by **strncat_s** in the array of **s1max** characters pointed to by **s1** take unspecified values when **strncat_s** returns.[26]

---

24) A zero return value implies that all of the requested characters from the string pointed to by **s2** were appended to the string pointed to by **s1** and that the result in **s1** is null terminated.

25) This means that **s1** was not null terminated upon entry to **strncat_s**.

6 If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

7 The **strncat_s** function returns zero[27] if there was no diagnosed undefined behavior. Otherwise, a non-zero value is returned.

8 EXAMPLE 1 The **strncat_s** function can be used to copy a string without the danger that the result will not be null terminated or that characters will be written past the end of the destination array.

```
#define __STDC_WANT_SECURE_LIB__ 1
#include <string.h>
/* … */
char s1[100] = "good";
char s2[6] = "hello";
char s3[6] = "hello";
char s4[7] = "abc";
char s5[1000] = "bye";
int r1, r2, r3, r4;
r1 = strncat_s(s1, 100, s5, 1000);
r2 = strncat_s(s2, 6, "", 1);
r3 = strncat_s(s3, 6, "X", 2);
r4 = strncat_s(s4, 7, "defghijklmn", 3);
```

After the first call **r1** will have the value zero and **s1** will contain the sequence **goodbye\0**.
After the second call **r2** will have the value zero and **s2** will contain the sequence **hello\0**.
After the third call **r3** will have the value **ERANGE** and **s3** will contain the sequence **\0**.
After the fourth call **r4** will have the value zero and **s4** will contain the sequence **abcdef\0**.

5.6.3 Search functions

5.6.3.1 The strtok_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <string.h>
char *strtok_s(char * restrict s1,
     const char * restrict s2,
     char ** restrict ptr);
```

---

26) This allows an implementation to append characters from **s2** to **s1** while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of **s1** before discovering that the first element should be set to the null character.

27) A zero return value implies that all of the requested characters from the string pointed to by **s2** were appended to the string pointed to by **s1** and that the result in **s1** is null terminated.

Description

2 If **s2 == NULL || ptr == NULL || (*ptr == null && s1 == NULL)**, there is diagnosed undefined behavior, and the **strtok_s** returns.

3 A sequence of calls to the **strtok_s** function breaks the string pointed to by **s1** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **s2**. The third argument points to a caller-provided **char** pointer into which the **strtok_s** function stores information necessary for it to continue scanning the same string.

4 The first call in a sequence has a non-null first argument and stores an initial value in the object pointed to by **ptr**. Subsequent calls in the sequence have a null first argument and the object pointed to by **ptr** is required to have the value stored by the previous call in the sequence, which is then updated. The separator string pointed to by **s2** may be different from call to call.

5 The first call in the sequence searches the string pointed to by **s1** for the first character that is *not* contained in the current separator string pointed to by **s2**. If no such character is found, then there are no tokens in the string pointed to by **s1** and the **strtok_s** function returns a null pointer. If such a character is found, it is the start of the first token.

6 The **strtok_s** function then searches from there for the first character in **s1** that *is* contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by **s1**, and subsequent searches in the same string for a token return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token.

7 In all cases, the **strtok_s** function stores sufficient information in the pointer pointed to by **ptr** so that subsequent calls, with a null pointer for **s1** and the unmodified pointer value for **ptr**, shall start searching just past the element overwritten by a null character (if any).

Returns

8 The **strtok_s** function returns a pointer to the first character of a token, or a null pointer if there is no token or there is diagnosed undefined behavior.

9 EXAMPLE

```
#define __STDC_WANT_SECURE_LIB__ 1
#include <string.h>
static char str1[] = "?a???b,,,#c";
static char str2[] = "\t \t";
char *t, *ptr1, *ptr2;
```

```
t = strtok_s(str1, "?", &ptr1);      // t points to the token "a"
t = strtok_s(NULL, ",", &ptr1);      // t points to the token "??b"
t = strtok_s(str2, " \t", &ptr2);    // t is a null pointer
t = strtok_s(NULL, "#,", &ptr1);     // t points to the token "c"
t = strtok_s(NULL, "?", &ptr1);      // t is a null pointer
```

5.6.4 Miscellaneous functions

5.6.4.1 The `strerror_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <string.h>
errno_t strerror_s(char *s, rsize_t maxsize,
     errno_t errnum);
```

Description

2 If **s** is a null pointer or **maxsize > RSIZE_MAX** or **maxsize** equals 0, then there is diagnosed undefined behavior, and the array (if any) pointed to by **s** is not modified.

3 The **strerror_s** function maps the number in **errnum** to a locale-specific message string. Typically, the values for **errnum** come from **errno**, but **strerror_s** shall map any value of type **int** to a message.

4 If the length of the desired string is less than **maxsize**, then the string is copied to the array pointed to by **s**.

5 Otherwise, if **maxsize** is greater than zero, then **maxsize-1** characters are copied from the string to the array pointed to by **s** and then **s[maxsize-1]** is set to the null character. Then, if **maxsize** is greater than 3, then **s[maxsize-2]**, **s[maxsize-3]**, and **s[maxsize-4]** are set to the character period (**.**).

Returns

6 The **strerror_s** function returns zero if the length of the desired string was less than **maxsize** and there was no diagnosed undefined behavior. Otherwise, the **strerror_s** function returns a non-zero value.

5.6.4.2 The `strnlen_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <string.h>
size_t strnlen_s(const char *s, size_t maxsize);
```

Description

2 The **strnlen_s** function computes the length of the string pointed to by **s**.

Returns

3 If **s** is a null pointer, then the **strnlen_s** function returns zero.

4 Otherwise, the **strnlen_s** function returns the number of characters that precede the terminating null character. If there is no null character in the first **maxsize** characters of **s** then strnlen_s returns **maxsize**. At most the first **maxsize** characters of **s** shall be accessed by **strnlen_s**.

5.7 Date and time <time.h>

1 The header **<time.h>** defines two types.

2 The types are

errno_t

which is type **int**; and

rsize_t

which is the type **size_t**.

5.7.1 Components of time

1 A broken-down time is *normalized* if the values of the members of the **tm** structure are in their normal rages.

5.7.2 Time conversion functions

1 Like the **strftime** function, the **asctime_s** and **ctime_s** functions do not return a pointer to a static object, and other library functions are permitted to call them.

5.7.2.1 The asctime_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <time.h>
errno_t asctime_s(char *s, rsize_t maxsize,
     const struct tm *timeptr);
```

Description

2 There is diagnosed undefined behavior if any of the following conditions are true:

— **s** or **timer** is a null pointer.

— **maxsize < 26 || maxsize >** RSIZE_MAX

— The broken-down time pointed to by **timeptr** is not normalized.

— The calendar year represented by the broken-down time pointed to by **timeptr** is less than year 0 or more than year 9999.

If there is diagnosed undefined behavior, there is no attempt to convert the time, and **s[0]** is set to a null character if **s != NULL && maxsize > 0 && maxsize <= RSIZE_MAX**.

3 The **asctime_s** function converts the normalized broken-down time in the structure pointed to by **timeptr** into a 26 character (including the null character) string in the

form

```
Sun Sep 16 01:03:52 1973\n\0
```

The fields making up this string are (in order):

1. The name of the day of the week represented by **`timeptr->tm_wday`** using the following three character weekday names: Sun, Mon, Tue, Wed, Thu, Fri, and Sat.
2. The character space.
3. The name of the month represented by **`timeptr->tm_mon`** using the following three character month names: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec.
4. The character space.
5. The value of **`timeptr->tm_mday`** as if printed using the **`fprintf`** format **`"%2d"`**.
6. The character space.
7. The value of **`timeptr->tm_hour`** as if printed using the **`fprintf`** format **`"%.2d"`**.
8. The character colon.
9. The value of **`timeptr->tm_min`** as if printed using the **`fprintf`** format **`"%.2d"`**.
10. The character colon.
11. The value of **`timeptr->tm_sec`** as if printed using the **`fprintf`** format **`"%.2d"`**.
12. The character space.
13. The value of **`timeptr->tm_year + 1900`** as if printed using the **`fprintf`** format **`"%4d"`**.
14. The character new line.
15. The null character.

Returns

4 The **`asctime_s`** function returns zero if the time was successfully converted and stored into the array pointed to by **`s`**. Otherwise, it returns a non-zero value.

5.7.2.2 The `ctime_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <time.h>
errno_t ctime_s(char *s, rsize_t maxsize,
     const time_t *timer);
```

Description

2 There is diagnosed undefined behavior if any of the following conditions are true:

— **s** or **timer** is a null pointer.

— **maxsize < 26 || maxsize > RSIZE_MAX**

If there is diagnosed undefined behavior, **s[0]** is set to a null character if **s != NULL && maxsize > 0 && maxsize <= RSIZE_MAX**.

3 The **ctime_s** function converts the calendar time pointed to by **timer** to local time in the form of a string. It is equivalent to

```
asctime_s(s, maxsize, localtime(timer))
```

Returns

4 The **ctime_s** function returns zero if the time was successfully converted and stored into the array pointed to by **s**. Otherwise, it returns a non-zero value.

5.7.2.3 The `gmtime_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <time.h>
struct tm *gmtime_s(const time_t * restrict timer,
     struct tm * restrict result);
```

Description

2 There is diagnosed undefined behavior if **timer** or **result** is a null pointer. If there is diagnosed undefined behavior, there is no attempt to convert the time.

3 The **gmtime_s** function converts the calendar time pointed to by **timer** into a broken-down time, expressed as UTC. The broken-down time is stored in the structure pointed to by **result**.

Returns

4 The **gmtime_s** function returns **result**, or a null pointer if the specified time cannot be converted to UTC or there is diagnosed undefined behavior.

5.7.2.4 The localtime_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <time.h>
struct tm *localtime_s(const time_t * restrict timer,
     struct tm * restrict result);
```

Description

2 There is diagnosed undefined behavior if **timer** or **result** is a null pointer. If there is diagnosed undefined behavior, there is no attempt to convert the time.

3 The **localtime_s** function converts the calendar time pointed to by **timer** into a broken-down time, expressed as local time. The broken-down time is stored in the structure pointed to by **result**.

Returns

4 The **localtime_s** function returns **result**, or a null pointer if the specified time cannot be converted to local time or there is diagnosed undefined behavior.

5.8 Extended multibyte and wide character utilities <wchar.h>

1 The header **<wchar.h>** defines two types.

2 The types are

errno_t

which is type **int**; and

rsize_t

which is the type **size_t**.

3 Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the objects take on unspecified values.

5.8.1 Formatted wide character input/output functions

5.8.1.1 The fwscanf_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdio.h>
#include <wchar.h>
int fwscanf_s(FILE * restrict stream,
     const wchar_t * restrict format, ...);
```

Description

2 If either **stream** or **format** is a null pointer, there is diagnosed undefined behavior, and the **fwscanf_s** function does not attempt to perform input.

3 The **fwscanf_s** function is equivalent to **fwscanf** except that the **c**, **s**, and **[** conversion specifiers apply to a pair of arguments (unless assignment suppression is indicated by a *****). The first of these arguments is the same as for **fwscanf**. That argument is immediately followed in the argument list by the second argument, which has type **size_t** and gives the number of elements in the array pointed to by the first argument of the pair. If the first argument points to a scalar object, it is considered to be an array of one element.[28)]

4 A matching failure occurs if the number of elements in a receiving object is insufficient to hold the converted input (including any trailing null character).

Returns

5 The **fwscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is diagnosed undefined behavior. Otherwise, the

fwscanf_s function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

5.8.1.2 The swscanf_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <wchar.h>
int swscanf_s(const wchar_t * restrict s,
     const wchar_t * restrict format, ...);
```

Description

2 If either **s** or **format** is a null pointer, there is diagnosed undefined behavior, and the **swscanf_s** function does not attempt to perform input.

3 The **swscanf_s** function is equivalent to **fwscanf_s**, except that the argument **s** specifies a wide string from which the input is to be obtained, rather than from a stream. Reaching the end of the wide string is equivalent to encountering end-of-file for the **fwscanf_s** function.

Returns

4 The **swscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is diagnosed undefined behavior. Otherwise, the **swscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

---

28) If the format is known at translation time, an implementation may issue a diagnostic for any argument used to store the result from a **c**, **s**, or **[** conversion specifier if that argument is not followed by an argument of a type compatible with **rsize_t**. A limited amount of checking may be done if even if the format is not known at translation time. For example, an implementation may issue a diagnostic for each argument after **format** that has of type pointer to one of **char**, **signed char**, **unsigned char**, or **void** that is not followed by an argument of a type compatible with **rsize_t**. The diagnostic could warn that unless the pointer is being used with a conversion specifier using the **hh** length modifier, a length argument must follow the pointer argument. Another useful diagnostic could flag any non-pointer argument following **format** that did not have a type compatible with **rsize_t**.

5.8.1.3 The `vfwscanf_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwscanf_s(FILE * restrict stream,
     const wchar_t * restrict format,
     va_list arg);
```

Description

2 If either **stream** or **format** is a null pointer, there is diagnosed undefined behavior, and the **vfwscanf_s** function does not attempt to perform input.

3 The **vfwscanf_s** function is equivalent to **fwscanf_s**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfwscanf_s** function does not invoke the **va_end** macro.[29)]

Returns

4 The **vfwscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is diagnosed undefined behavior. Otherwise, the **vfwscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

5.8.1.4 The `vswscanf_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdarg.h>
#include <wchar.h>
int vswscanf_s(const wchar_t * restrict s,
     const wchar_t * restrict format,
     va_list arg);
```

Description

2 If either **s** or **format** is a null pointer, there is diagnosed undefined behavior, and the **fscanf_s** function does not attempt to perform input.

---

29) As the functions **vfwscanf_s**, **vwscanf_s**, and **vswscanf_s** invoke the **va_arg** macro, the value of **arg** after the return is indeterminate.

3 The **vswscanf_s** function is equivalent to **swscanf_s**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vswscanf_s** function does not invoke the **va_end** macro.[29)]

Returns

4 The **vswscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is diagnosed undefined behavior. Otherwise, the **vswscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

5.8.1.5 The vwscanf_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdarg.h>
#include <wchar.h>
int vwscanf_s(const wchar_t * restrict format,
     va_list arg);
```

Description

2 If **format** is a null pointer, there is diagnosed undefined behavior, and the **vwscanf_s** function does not attempt to perform input.

3 The **vwscanf_s** function is equivalent to **wscanf_s**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vwscanf_s** function does not invoke the **va_end** macro.[29)]

Returns

4 The **vwscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is diagnosed undefined behavior. Otherwise, the **vwscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

5.8.1.6 The wscanf_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <wchar.h>
int wscanf_s(const wchar_t * restrict format, ...);
```

Description

2 If **format** is a null pointer, there is diagnosed undefined behavior, and the **wscanf_s** function does not attempt to perform input.

3 The **wscanf_s** function is equivalent to **fwscanf_s** with the argument **stdin** interposed before the arguments to **wscanf_s**.

Returns

4 The **wscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is diagnosed undefined behavior. Otherwise, the **wscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

5.8.2 General wide string utilities

5.8.2.1 Wide string copying functions

5.8.2.1.1 The wcscpy_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <wchar.h>
errno_t wcscpy_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2);
```

Description

2 There is diagnosed undefined behavior if any of the following conditions are true:

— **s1 == NULL || s2 == NULL**

— **s1max > RSIZE_MAX**

— **s1max == 0**

— **s1max <= wcsnlen_s(s2, s1max)**

If there is diagnosed undefined behavior, then **s1[0]** is set to the null wide character if **s1 != NULL && s1max > 0 && s1max <= RSIZE_MAX**.

3 Otherwise, the wide characters pointed to by **s2** up to and including the null wide character are copied to the array pointed to by **s1**.

4 All elements following the terminating null wide character (if any) written by **wcscpy_s** in the array of **s1max** wide characters pointed to by **s1** take unspecified values when **wcscpy_s** returns.[30)]

Returns

5 The **wcscpy_s** function returns zero[31] if there was no diagnosed undefined behavior. Otherwise, a non-zero value is returned.

5.8.2.1.2 The wcsncpy_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <wchar.h>
errno_t wcsncpy_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2,
     rsize_t n);
```

Description

2 There is diagnosed undefined behavior if any of the following conditions are true:

— **s1 == NULL || s2 == NULL**

— **s1max > RSIZE_MAX || n > RSIZE_MAX**,

— **s1max == 0**

— **n >= s1max && s1max <= wcsnlen_s(s2, s1max)**

If there is diagnosed undefined behavior, then **s1[0]** is set to the null wide character if **s1 != NULL && s1max > 0 && s1max <= RSIZE_MAX**.

3 Otherwise, the **wcsncpy_s** function copies not more than **n** successive wide characters (wide characters that follow a null wide character are not copied) from the array pointed to by **s2** to the array pointed to by **s1**. If no null wide character was copied from **s2**, then **s1[n]** is set to a null wide character.

4 All elements following the terminating null wide character (if any) written by **wcsncpy_s** in the array of **s1max** wide characters pointed to by **s1** take unspecified values when **wcsncpy_s** returns.[32]

---

30) This allows an implementation to copy wide characters from **s2** to **s1** while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of **s1** before discovering that the first element should be set to the null wide character.

31) A zero return value implies that all of the requested wide characters from the string pointed to by **s2** fit within the array pointed to by **s1** and that the result in **s1** is null terminated.

32) This allows an implementation to copy wide characters from **s2** to **s1** while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of **s1** before discovering that the first element should be set to the null wide character.

Returns

5 The **wcsncpy_s** function returns zero[33)] if there was no diagnosed undefined behavior. Otherwise, a non-zero value is returned.

6 EXAMPLE 1 The **wcsncpy_s** function can be used to copy a wide string without the danger that the result will not be null terminated or that wide characters will be written past the end of the destination array.

```
#define __STDC_WANT_SECURE_LIB__ 1
#include <wchar.h>
/* ... */
wchar_t src1[100] = L"hello";
wchar_t src2[7] = {L'g', L'o', L'o', L'd', L'b', L'y', L'e'};
wchar_t dst1[6], dst2[5], dst3[5];
int r1, r2, r3;
r1 = wcsncpy_s(dst1, 6, src1, 100);
r2 = wcsncpy_s(dst2, 5, src2, 7);
r3 = wcsncpy_s(dst3, 5, src2, 4);
```

The first call will assign to **r1** the value zero and to **dst1** the sequence of wide characters **hello\0**. The second call will assign to **r2** the value **ERANGE** and to **dst2** the sequence of wide characters **\0**. The third call will assign to **r3** the value zero and to **dst3** the sequence of wide characters **good\0**.

5.8.2.1.3 The wmemcpy_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <wchar.h>
errno_t wmemcpy_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2,
     rsize_t n);
```

Description

2 There is diagnosed undefined behavior if any of the following conditions are true:

— **s1 == NULL || s2 == NULL**

— **s1max > RSIZE_MAX || n > RSIZE_MAX**

— **n > s1max**

If there is diagnosed undefined behavior, the **memcpy_s** function stores zeros in the first **s1max** wide characters of the object pointed to by **s1** if **s1 != NULL && s1max <= RSIZE_MAX**

33) A zero return value implies that all of the requested wide characters from the string pointed to by **s2** fit within the array pointed to by **s1** and that the result in **s1** is null terminated.

3 Otherwise, the **wmemcpy_s** function copies **n** successive wide characters from the object pointed to by **s2** into the object pointed to by **s1**.

Returns

4 The **wmemcpy_s** function returns zero if there was no diagnosed undefined behavior. Otherwise, a non-zero value is returned.

5.8.2.1.4 The wmemmove_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <wchar.h>
errno_t wmemmove_s(wchar_t *s1, rsize_t s1max,
     const wchar_t *s2, rsize_t n);
```

Description

2 There is diagnosed undefined behavior if any of the following conditions are true:

— **s1 == NULL || s2 == NULL**

— **s1max > RSIZE_MAX || n > RSIZE_MAX**

— **n > s1max**

If there is diagnosed undefined behavior, the **memmove_s** function stores zeros in the first **s1max** wide characters of the object pointed to by **s1** if **s1 != NULL && s1max <= RSIZE_MAX**

3 Otherwise, the **wmemmove_s** function copies **n** successive wide characters from the object pointed to by **s2** into the object pointed to by **s1**. This copying takes place as if the **n** wide characters from the object pointed to by **s2** are first copied into a temporary array of **n** wide characters that does not overlap the objects pointed to by **s1** or **s2**, and then the **n** wide characters from the temporary array are copied into the object pointed to by **s1**.

Returns

4 The **wmemmove_s** function returns zero if there was no diagnosed undefined behavior. Otherwise, a non-zero value is returned.

5.8.2.2 Wide string concatenation functions

5.8.2.2.1 The `wcscat_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <wchar.h>
errno_t wcscat_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2);
```

Description

2 Let, *m* have the value **s1max - wcsnlen_s(s1, s1max)** upon entry to **wcscat_s.**

3 There is diagnosed undefined behavior if any of the following conditions are true:

— **s1 == NULL || s2 == NULL**

— **s1max > RSIZE_MAX**

— **s1max == 0**

— *m* **== 0**[34)]

— *m* **<= wcsnlen(s2,** *m***)**,

If there is diagnosed undefined behavior, then **s1[0]** is set to the null wide character if **s1 != NULL && s1max > 0 && s1max <= RSIZE_MAX**.

4 Otherwise, the wide characters pointed to by **s2** up to and including the null wide character are appended to the end of the wide string pointed to by **s1**. The initial wide character from **s2** overwrites the null wide character at the end of **s1**.

5 All elements following the terminating null wide character (if any) written by **wcscat_s** in the array of **s1max** wide characters pointed to by **s1** take unspecified values when **wcscat_s** returns.[35)] *

Returns

6 The **wcscat_s** function returns zero[36)] if there was no diagnosed undefined behavior.

---

34) This means that **s1** was not null terminated upon entry to **wcscat_s**.

35) This allows an implementation to append wide characters from **s2** to **s1** while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of **s1** before discovering that the first element should be set to the null wide character.

36) A zero return value implies that all of the requested wide characters from the wide string pointed to by **s2** were appended to the wide string pointed to by **s1** and that the result in **s1** is null terminated.

Otherwise, a non-zero value is returned.

5.8.2.2.2 The `wcsncat_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <wchar.h>
errno_t wcsncat_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2,
     rsize_t n);
```

Description

2 Let, *m* have the value **s1max - wcsnlen_s(s1, s1max)** upon entry to **wcsncat_s.**

3 There is diagnosed undefined behavior if any of the following conditions are true:

— **s1 == NULL || s2 == NULL**

— **s1max >** **RSIZE_MAX** **|| n >** **RSIZE_MAX**,

— **s1max == 0**

— *m* **== 0**[37)]

— **n >=** *m* **&&** *m* **<= strnlen_s(s2,** *m***)**,

If there is diagnosed undefined behavior, then **s1[0]** is set to the null wide character if **s1 != NULL && s1max > 0 && s1max <= RSIZE_MAX**.

4 Otherwise, the **wcsncat_s** function appends not more than **n** successive wide characters (wide characters that follow a null wide character are not copied) from the array pointed to by **s2** to the end of the wide string pointed to by **s1**. The initial wide character from **s2** overwrites the null wide character at the end of **s1**. If no null wide character was copied from **s2**, then **s1[s1max-m+n]** is set to a null wide character.

5 All elements following the terminating null wide character (if any) written by **wcsncat_s** in the array of **s1max** wide characters pointed to by **s1** take unspecified values when **wcsncat_s** returns.[38)]

---

37) This means that **s1** was not null terminated upon entry to **wcsncat_s**.

38) This allows an implementation to append wide characters from **s2** to **s1** while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of **s1** before discovering that the first element should be set to the null wide character.

Returns

6 The **wcsncat_s** function returns zero[39] if there was no diagnosed undefined behavior. Otherwise, a non-zero value is returned.

7 EXAMPLE 1 The **wcsncat_s** function can be used to copy a wide string without the danger that the result will not be null terminated or that wide characters will be written past the end of the destination array.

```
#define __STDC_WANT_SECURE_LIB__ 1
#include <wchar.h>
/* … */
wchar_t s1[100] = L"good";
wchar_t s2[6] = L"hello";
wchar_t s3[6] = L"hello";
wchar_t s4[7] = L"abc";
wchar_t s5[1000] = L"bye";
int r1, r2, r3, r4;
r1 = wcsncat_s(s1, 100, s5, 1000);
r2 = wcsncat_s(s2, 6, L"", 1);
r3 = wcsncat_s(s3, 6, L"X", 2);
r4 = wcsncat_s(s4, 7, L"defghijklmn", 3);
```

After the first call **r1** will have the value zero and **s1** will be the wide character sequence **goodbye\0**.
After the second call **r2** will have the value zero and **s2** will be the wide character sequence **hello\0**.
After the third call **r3** will have the value **ERANGE** and **s3** will be the wide character sequence **\0**.
After the fourth call **r4** will have the value zero and **s4** will be the wide character sequence **abcdef\0**.

5.8.2.3 Miscellaneous functions

5.8.2.3.1 The wcsnlen_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <wchar.h>
size_t wcsnlen_s(const wchar_t *s, size_t maxsize);
```

Description

2 The **wcsnlen_s** function computes the length of the wide string pointed to by **s**.

Returns

3 If **s** is a null pointer, then the **wcsnlen_s** function returns zero.

4 Otherwise, the **wcsnlen_s** function returns the number of wide characters that precede the terminating null wide character. If there is no null wide character in the first **maxsize** wide characters of **s** then wcsnlen_s returns **maxsize**. At most the first

---

39) A zero return value implies that all of the requested wide characters from the wide string pointed to by **s2** were appended to the wide string pointed to by **s1** and that the result in **s1** is null terminated.

`maxsize` wide characters of **`s`** shall be accessed by **`wcsnlen_s`**.

Index