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.

7 The remaining feature of this technical report is a new random number generator that is suitable for use in cryptography.

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 3 of this Technical Report should be read as if it was merged into Subclause 6.10.8 of ISO/IEC 9899:1999. Clause 4 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. Predefined macro names

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

`__STDC_SECURE_LIB__` The integer constant **`200410L`**, 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.

4. Library

4.1 Introduction

4.1.1 Standard headers

1 The functions, macros, and types defined in Clause 4 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 4 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.[2)]

3 It is implementation-defined whether the functions, macros, and types defined in Clause 4 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.[3)]

4 If a given standard header is included more than once in a given scope, then it is undefined behavior if **`__STDC_WANT_SECURE_LIB__`** is defined differently for some of those inclusions.

4.1.2 Use of errno

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

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

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 4 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.

4.2 Errors `<errno.h>`

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

2 The type is

    errno_t

which is type **`int`**.[4)]

4) 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`**.

4.3 Input/output `<stdio.h>`

1 The header **`<stdio.h>`** defines several macros and one type. |

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 type is

`errno_t`

which is type **`int`**.

4.3.1 Operations on files

4.3.1.1 The `tmpnam_s` function

Synopsis

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

Description

2 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.[5] 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.

5) 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.

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

4 The implementation shall behave as if no library function except **tmpnam** calls the **tmpnam_s** function.[6)]

Returns

5 If no suitable string can be generated, or if the length of the string is not less than the value of **maxsize**, the **tmpnam_s** function writes a null character to **s[0]** (only if **maxsize** is greater than zero) and returns **ERANGE**.

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

Environmental limits

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

4.3.2 Formatted input/output functions

4.3.2.1 The fscanf function

1 The **fscanf** function[7)] now allows an optional precision in conversion specifications. The precision can be used to prevent reading more data into an array than it can store.

2 The definition of conversion specification in Paragraph 3 of Subclause 7.19.6.2 of ISO/IEC 9899:1999 is modified as follows:

3 Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

— An optional assignment-suppressing character *****.

— An optional nonzero decimal integer that specifies the maximum field width (in characters).

— An optional *precision* that gives the number of elements in the array to receive the converted input for the **c**, **s**, and **[** conversions. For the purpose of precision, a scalar object receiving converted input is considered to be an array of one element. The precision takes the form of a period (**.**) followed either by an asterisk ***** (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

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

7) This change also affects all functions defined in terms of an equivalence to **fscanf**.

— An optional *length modifier* that specifies the size of the receiving object.

— A *conversion specifier* character that specifies the type of conversion to be applied.

4 As noted above, a precision may be indicated by an asterisk. In this case, an argument of type **`size_t`** supplies the precision.[8] The argument specifying precision shall appear before the argument (if any) to receive the converted input. A negative precision argument is taken as if the precision were zero.

5 If a **`c`**, **`s`**, or **`[`** conversion lacks a precision, the precision is taken as **`(size_t)(-1)`**.[9]

6 Paragraph 10 of Subclause 7.19.6.2 of ISO/IEC 9899:1999 is changed to:

7 Except in the case of a **`%`** specifier, the input item (or, in the case of a **`%n`** directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. The directive also fails with a matching failure if the conversion result (including any trailing null character) for a **`c`**, **`s`**, or **`[`** specifier does not fit within the number of elements given by the precision. Unless assignment suppression was indicated by a **`*`**, the conversion result is placed in the object pointed to by the first argument following the **`format`** argument that has not already been used as a precision or received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

4.3.2.2 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 The **`fscanf_s`** function is equivalent to **`fscanf`** except that if a **`c`**, **`s`**, or **`[`** conversion lacks a precision, the precision is taken as zero.[10] *

8) If the format is known at translation time, an implementation may issue a diagnostic if the function argument corresponding to the asterisk does not have type **`size_t`**.

9) In effect, this causes **`fscanf`** to assume that the converted input always fits within the receiving array. In contrast, a precision of zero causes **`fscanf`** to assume the converted input never fits.

10) Thus, a precision must be explicitly specified to avoid these conversions always failing. If the format is known at translation time, an implementation may issue a diagnostic for any **`c`**, **`s`**, or **`[`** conversion that lacks a precision. If the precision is specified with an asterisk, the implementation may issue a diagnostic if the corresponding function argument does not have type **`size_t`**.

Returns

3 The **fscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

4 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%.50s", &i, &x, name);
```

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**.

5 EXAMPLE 2 The call:

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

4.3.2.3 The scanf_s function

Synopsis

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

Description

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

Returns

3 The **scanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

4.3.2.4 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 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 behavior is undefined.

Returns

3 The **`sscanf_s`** function returns the value of the macro **`EOF`** if an input failure occurs before any conversion. 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.

4.3.2.5 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 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.[11)]

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

Returns

3 The **vfscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

4.3.2.6 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 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.[11)]

Returns

3 The **vscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

4.3.2.7 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);
```

Description

2 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.[11)]

Returns

3 The **vsscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

4.3.3 Character input/output functions

4.3.3.1 The gets_s function

Synopsis

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

Description

2 If **n** is equal to zero or **s** is a null pointer, then 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 **n** is equal to zero, or if **s** is a null pointer, 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.

4.4 General utilities `<stdlib.h>`

1 The header **`<stdlib.h>`** defines a macro and a type.

2 The macro is

```
RAND_MAX_S
```

which expands to an integer constant expression that is the maximum value returned by the **`rand_s`** function.

3 The type is

```
errno_t
```

which is type **`int`**.

4.4.1 Pseudo-random sequence generation functions

4.4.1.1 The `rand_s` function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <stdlib.h>
int rand_s(void);
```

Description

2 The **`rand_s`** function computes a sequence of pseudo-random integers in the range 0 to **`RAND_MAX_S`**.

3 These random numbers are generated using methods appropriate for use in cryptography.

Returns

4 The **`rand_s`** function returns a pseudo-random integer.

Environmental limits

5 The value of the **`RAND_MAX_S`** macro shall be at least 32767.

4.4.2 Communication with the environment

4.4.2.1 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, size_t maxsize,
          const char * restrict name);
```

Description

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

3 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`**.

4 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.

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

Returns

6 The **`getenv_s`** function returns zero if the specified **`name`** is found and the length of the associated string is less than **`maxsize`**. Otherwise, **`ERANGE`** is returned.

4.4.3 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.12) 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.

4.4.3.1 The bsearch_s function

Synopsis

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

Description

2 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**.

3 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.13) 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

12) 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
```

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

function.[14]

Returns

4 The **`bsearch_s`** function returns a pointer to a matching element of the array, or a null pointer if no match is found. If two elements compare as equal, which element is matched is unspecified.

4.4.3.2 The `qsort_s` function

Synopsis

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

Description

2 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`**.

3 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.[14]

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

Returns

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

14) 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.

4.5 String handling `<string.h>`

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

2 The type is

```
errno_t
```

which is type **`int`**.

4.5.1 Copying functions

4.5.1.1 The `memcpy_s` function

Synopsis

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

Description

2 If **`s1`** or **`s2`** is a null pointer, then no copying is performed.

3 Otherwise, if **`n`** is less than or equal to **`s1max`**, the **`memcpy_s`** function copies **`n`** characters from the object pointed to by **`s2`** into the object pointed to by **`s1`**. Otherwise, the **`memcpy_s`** function stores zeros in the first **`s1max`** characters of the object pointed to by **`s1`**.

4 If copying takes place between objects that overlap, the behavior is undefined.

Returns

5 The **`memcpy_s`** function returns zero if **`n`** is less than or equal to **`s1max`** and **`s1`** and **`s2`** are not null pointers. Otherwise, **`ERANGE`** is returned.

4.5.1.2 The `memmove_s` function

Synopsis

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

Description

2 If **`s1`** or **`s2`** is a null pointer, then no copying is performed.

3 Otherwise, if **`n`** is less than or equal to **`s1max`**, 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`**.

4 Otherwise, if **`n`** is greater than **`s1max`**, the **`memmove_s`** function stores zeros in the first **`s1max`** characters of the object pointed to by **`s1`**. |

Returns

5 The **`memmove_s`** function returns zero if **`n`** is less than or equal to **`s1max`** and **`s1`** and **`s2`** are not null pointers. Otherwise, **`ERANGE`** is returned. |

4.5.1.3 The `strcpy_s` function

Synopsis

1

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

|

Description

2 If **`s1`** or **`s2`** is a null pointer, or if **`s1max`** is equal to zero, then no copying is performed. |

3 Otherwise, if **`s1max`** is greater than **`strnlen_s(s2, s1max)`**, then the characters pointed to by **`s2`** up to and including the null character are copied to the array pointed to by **`s1`**. |

4 Otherwise, **`s1[0]`** is set to the null character.

5 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.[15)]

6 If copying takes place between objects that overlap, the behavior is undefined.

Returns

7 The **`strcpy_s`** function returns **`ERANGE`** if **`s1`** or **`s2`** is a null pointer, or if **`s1max`** equals zero, or if **`strnlen_s(s2, s1max)`** is equal to **`s1max`**. Otherwise, zero is returned.[16)] |
|

---

15) 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.

4.5.1.4 The `strncpy_s` function

Synopsis

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

Description

2 If **`s1`** or **`s2`** is a null pointer, or if **`s1max`** is equal to zero, then no copying is performed.

3 Otherwise, if **`n`** is greater than or equal to **`s1max`**, then the behavior of the **`strncpy_s`** function depends upon whether there is a null character in the first **`s1max`** characters of the array pointed to by **`s2`**. If there is a null character, then the characters pointed to by **`s2`** up to and including the null character are copied to the array pointed to by **`s1`**. If there is no null character, then **`s1[0]`** is set to the null character.

4 Otherwise, if **`n`** is less than **`s1max`**, then 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.

5 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.17)

6 If copying takes place between objects that overlap, the behavior is undefined.

Returns

7 The **`strncpy_s`** function returns **`ERANGE`** if **`s1`** or **`s2`** is a null pointer, or if **`s1max`** equals zero, or if **`n`** is greater than or equal to **`s1max`** and there is no null character in the first **`s1max`** characters of **`s2`**. Otherwise, zero is returned.18)

---

16) 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.

17) 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.

18) 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.

8 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**.

4.5.2 Concatenation functions

4.5.2.1 The strcat_s function

Synopsis

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

Description

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

3 If *m* is equal to zero,[19] or if **s1** or **s2** is a null pointer, then no copying is performed.

4 Otherwise, if *m* is greater than **strnlen_s(s2,** *m***)**, then 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 Otherwise **s1[0]** is set to the null character.

6 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.[20]

7 If copying takes place between objects that overlap, the behavior is undefined.

Returns

8 The **strcat_s** function returns **ERANGE** if **s1** or **s2** is a null pointer, or if *m* equals zero, or if **strnlen_s(s2,** *m***)** is equal to *m*. Otherwise, zero is returned.[21]

4.5.2.2 The `strncat_s` function

Synopsis

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

Description

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

3 If *m* is equal to zero,[22] or if **`s1`** or **`s2`** is a null pointer, then no copying is performed.

4 Otherwise, if **`n`** is greater than or equal to *m*, then the behavior of the **`strncat_s`** function depends upon whether there is a null character in the first *m* characters of the array pointed to by **`s2`**. If there is a null character, then 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`**. If there is no null character in the first *m* characters of the array pointed **`s2`** then **`s1[0]`** is set to the null character.

5 Otherwise, if **`n`** is less than *m*, then 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.

6 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.[23]

7 If copying takes place between objects that overlap, the behavior is undefined.

21) 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.

22) This means that **`s1`** was not null terminated upon entry to **`strncat_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

8 The **`strncat_s`** function returns **`ERANGE`** if **`s1`** or **`s2`** is a null pointer, or if *m* equals zero, or if **`n`** is greater than or equal to *m* and there is no null character in the first *m* characters of **`s2`**. Otherwise, zero is returned.[24)]

9 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`**.

4.5.3 Search functions

4.5.3.1 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);
```

Description

2 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.

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.

3 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.

4 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.

5 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.

6 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

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

8 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
```

4.5.4 Miscellaneous functions

4.5.4.1 The `strerror_s` function

Synopsis

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

Description

2 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.

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

4 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

5 The **`strerror_s`** function returns **`ERANGE`** if the length of the desired string was greater than or equal to **`maxsize`**. Otherwise, the **`strerror_s`** function returns zero.

4.5.4.2 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`**.

4.6 Date and time `<time.h>`

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

2 The type is

```
errno_t
```

which is type **`int`**.

4.6.1 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.

4.6.1.1 The `asctime_s` function

Synopsis

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

Description

2 The **`asctime_s`** function converts the broken-down time in the structure pointed to by **`timeptr`** into a string in the form

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

using the equivalent of the following algorithm.

```
errno_t asctime_s(char *s, size_t maxsize,
     const struct tm *timeptr)
{
     static const char wday_name[7][3] = {
          "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
     };
     static const char mon_name[12][3] = {
          "Jan", "Feb", "Mar", "Apr", "May", "Jun",
          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
     };
     int n;
     struct tm tmptime = *timeptr;
     int year;

     mktime(&tmptime);
     year = 1900 + tmptime.tm_year;
     if (year > 9999)
          year = 9999;
     n = snprintf(s, maxsize,
          "%.3s %.3s%3d %.2d:%.2d:%.2d %4d\n",
          wday_name[tmptime.tm_wday],
          mon_name[tmptime.tm_mon],
          tmptime.tm_mday, tmptime.tm_hour,
          tmptime.tm_min, tmptime.tm_sec,
          year);

     if (n < 0 || n >= maxsize) {
          s[0] = '\0';
          return ERANGE;
     }

     return 0;
}
```

Returns

3 The **`asctime_s`** function returns zero if the converted time fits within the first **`maxsize`** elements of the array pointed to by **`s`**.[25] Otherwise, it returns **`ERANGE`**.

25) A 26 character array is sufficient.

4.6.1.2 The `ctime_s` function

Synopsis

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

Description

2 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

3 The **`ctime_s`** function returns zero if the converted time fits within the first **`maxsize`** elements of the array pointed to by **`s`**.[26] Otherwise, it returns **`ERANGE`**.

4.6.1.3 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 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

3 The **`gmtime_s`** function returns **`result`**, or a null pointer if the specified time cannot be converted to UTC.

26) A 26 character array is sufficient for all times with four-digit years.

4.6.1.4 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 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

3 The **`localtime_s`** function returns **`result`**, or a null pointer if the specified time cannot be converted to local time. *

4.7 Extended multibyte and wide character utilities `<wchar.h>`

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

2 The type is

```
errno_t
```

which is type **`int`**.

4.7.1 Formatted wide character input/output functions

4.7.1.1 The `fwscanf` function

1 The **`fwscanf`** function[27] now allows an optional precision in conversion specifications. The precision can be used to prevent reading more data into an array than it can store.

2 The definition of conversion specification in Paragraph 3 of Subclause 7.24.2.2 of ISO/IEC 9899:1999 is modified as follows:

3 Each conversion specification is introduced by the wide character **`%`**. After the **`%`**, the following appear in sequence:

- An optional assignment-suppressing wide character **`*`**.
- An optional nonzero decimal integer that specifies the maximum field width (in wide characters).
- An optional *precision* that gives the number of elements in the array to receive the converted input for the **`c`**, **`s`**, and **`[`** conversions. For the purpose of precision, a scalar object receiving converted input is considered to be an array of one element. The precision takes the form of a period (**`.`**) followed either by an asterisk **`*`** (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional *length modifier* that specifies the size of the receiving object.
- A *conversion specifier* wide character that specifies the type of conversion to be applied.

4 As noted above, a precision may be indicated by an asterisk. In this case, an argument of type **`size_t`** supplies the precision.[28] The argument specifying precision shall appear before the argument (if any) to receive the converted input. A negative precision argument is taken as if the precision were zero.

27) This change also affects all functions defined in terms of an equivalence to **`fwscanf`**.

28) If the format is known at translation time, an implementation may issue a diagnostic if the function argument corresponding to the asterisk does not have type **`size_t`**.

5 If a **c**, **s**, or **[** conversion lacks a precision, the precision is taken as **(size_t)(-1)**.[29]

6 Paragraph 10 of Subclause 7.24.2.2 of ISO/IEC 9899:1999 is changed to:

7 Except in the case of a **%** specifier, the input item (or, in the case of a **%n** directive, the count of input wide characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. The directive also fails with a matching failure if the conversion result (including any trailing null character) for a **c**, **s**, or **[** specifier does not fit within the number of elements given by the precision. Unless assignment suppression was indicated by a *****, the conversion result is placed in the object pointed to by the first argument following the **format** argument that has not already been used as a precision or received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

4.7.1.2 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 The **fwscanf_s** function is equivalent to **fwscanf** except that if a **c**, **s**, or **[** conversion lacks a precision, the precision is taken as zero.[30]

Returns

3 The **fwscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

29) In effect, this causes **fwscanf** to assume that the converted input always fits within the receiving array. In contrast, a precision of zero causes **fwscanf** to assume the converted input never fits.

30) Thus, a precision must be explicitly specified to avoid these conversions always failing. If the format is known at translation time, an implementation may issue a diagnostic for any **c**, **s**, or **[** conversion that lacks a precision. If the precision is specified with an asterisk, the implementation may issue a diagnostic if the corresponding function argument does not have type **size_t**.

4.7.1.3 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 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

3 The **`swscanf_s`** function returns the value of the macro **`EOF`** if an input failure occurs before any conversion. 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.

4.7.1.4 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 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.[31)]

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

Returns

3 The **`vfwscanf_s`** function returns the value of the macro **`EOF`** if an input failure occurs before any conversion. 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.

4.7.1.5 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 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.[31)]

Returns

3 The **`vswscanf_s`** function returns the value of the macro **`EOF`** if an input failure occurs before any conversion. 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.

4.7.1.6 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 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.[31)]

Returns

3 The **`vwscanf_s`** function returns the value of the macro **`EOF`** if an input failure occurs before any conversion. 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.

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

Returns

3 The **`wscanf_s`** function returns the value of the macro **`EOF`** if an input failure occurs before any conversion. 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.

4.7.2 General wide string utilities

4.7.2.1 Wide string copying functions

4.7.2.1.1 The `wcscpy_s` function

Synopsis

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

Description

2 If **`s1`** or **`s2`** is a null pointer, or if **`s1max`** is equal to zero, then no copying is performed.

3 Otherwise, if **`s1max`** is greater than **`wcsnlen_s(s2, s1max)`**, then the characters pointed to by **`s2`** up to and including the null wide character are copied to the array pointed to by **`s1`**.

4 Otherwise, **`s1[0]`** is set to the null wide character.

5 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.[32)]

Returns

6 The **`wcscpy_s`** function returns **`ERANGE`** if **`s1`** or **`s2`** is a null pointer, or if **`s1max`** equals zero, or if **`wcsnlen_s(s2, s1max)`** is equal to **`s1max`**. Otherwise, zero is returned.[33)]

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.

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

4.7.2.1.2 The `wcsncpy_s` function

Synopsis

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

Description

2 If `s1` or `s2` is a null pointer, or if `s1max` is equal to zero, then no copying is performed.

3 Otherwise, if `n` is greater than or equal to `s1max`, then the behavior of the `wcsncpy_s` function depends upon whether there is a null wide character in the first `s1max` wide characters of the array pointed to by `s2`. If there is a null wide character, then the wide characters pointed to by `s2` up to and including the null wide character are copied to the array pointed to by `s1`. If there is no null wide character, then `s1[0]` is set to the null wide character.

4 Otherwise, if `n` is less than `s1max`, then 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.

5 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.[34)]

Returns

6 The `wcsncpy_s` function returns `ERANGE` if `s1` or `s2` is a null pointer, or if `s1max` equals zero, or if `n` is greater than or equal to `s1max` and there is no null wide character in the first `s1max` wide characters of `s2`. Otherwise, zero is returned.[35)]

7 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.

34) 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.

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

```
#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**.

4.7.2.1.3 The wmemcpy_s function

Synopsis

1

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

Description

2 If **s1** or **s2** is a null pointer, then no copying is performed.

3 Otherwise, if **n** is less than or equal to **s1max**, the **wmemcpy_s** function copies **n** successive wide characters from the object pointed to by **s2** into the object pointed to by **s1**. Otherwise, the **wmemcpy_s** function stores zeros in the first **s1max** wide characters of the object pointed to by **s1**.

Returns

4 The **wmemcpy_s** function returns zero if **n** is less than or equal to **s1max** and **s1** and **s2** are not null pointers. Otherwise, **ERANGE** is returned.

4.7.2.1.4 The wmemmove_s function

Synopsis

1

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

Description

2 If **s1** or **s2** is a null pointer, then no copying is performed.

3 Otherwise, if

4 If **n** is less than or equal to **s1max**, 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**.

5 If **n** is greater than **s1max**, the **wmemmove_s** function stores zeros in the first **s1max** wide characters of the object pointed to by **s1**.

Returns

6 The **wmemmove_s** function returns zero if **n** is less than or equal to **s1max** and **s1** and **s2** are not null pointers. Otherwise, **ERANGE** is returned.

4.7.2.2 Wide string concatenation functions

4.7.2.2.1 The wcscat_s function

Synopsis

1
```
#define __STDC_WANT_SECURE_LIB__ 1
#include <wchar.h>
errno_t wcscat_s(wchar_t * restrict s1,
     size_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 If *m* is equal to zero,[36] or if **s1** or **s2** is a null pointer, then no copying is performed.

4 Otherwise, if *m* is greater than **wcsnlen_s(s2,** *m***)**, then 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 Otherwise **s1[0]** is set to the null wide character.

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

6 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.[37)]

Returns

7 The **`wcscat_s`** function returns **`ERANGE`** if **`s1`** or **`s2`** is a null pointer, or if *m* equals zero, or if **`wcsnlen_s(s2,`** *m***`)`** is equal to *m*. Otherwise, zero is returned.[38)]

4.7.2.2.2 The `wcsncat_s` function

Synopsis

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

Description

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

3 If *m* is equal to zero,[39)] or if **`s1`** or **`s2`** is a null pointer, then no copying is performed.

4 Otherwise, if **`n`** is greater than or equal to *m*, then the behavior of the **`wcsncat_s`** function depends upon whether there is a null wide character in the first *m* wide characters of the array pointed to by **`s2`**. If there is a null wide character, then 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`**. If there is no null wide character in the first *m* wide characters of the array pointed **`s2`** then **`s1[0]`** is set to the null wide character.

5 Otherwise, if **`n`** is less than *m*, then 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`**.

37) 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.

38) 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.

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

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.

6 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.[40)]

Returns

7 The **wcsncat_s** function returns **ERANGE** if **s1** or **s2** is a null pointer, or if *m* equals zero, or if **n** is greater than or equal to *m* and there is no null wide character in the first *m* wide characters of **s2**. Otherwise, zero is returned.[41)]

8 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**.

40) 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.

41) 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.

4.7.2.3 Miscellaneous functions

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