SC22 WG14 N1173

Rationale for TR 24731
Extensions to the C Library
Part I: Bounds-checking interfaces

1 Introduction

November 3, 1988 the Internet was mentioned for one of the first times in news broadcasts. Most laymen had never heard of the Internet, which at the time (only!) consisted of around 60,000 computers, most of them enterprise or departmental sized machines, and technical workstations. The news story did not show the Internet is a very favorable light: the day before, for the first time, the Internet was under a large scale attack. Estimates vary, but as many as 10% of the computers on the Internet had been infected by a worm program.

The Morris worm, or Great Internet Worm, was able to infect computers with different instruction sets from different manufacturers running different versions of the UNIX operating system. The worm was able to infect systems remotely over the network without an account or privileges on the target machine. The worm program used multiple different attacks (including a password dictionary attack), but one of its attacks would become a model for many attacks against many different operating systems in the future. This form of attack has become so common that the design of some recent microprocessors include a “no execute” memory page bit to mitigate the attack.

The attack is the buffer overrun. This attack can be used anytime a program writes past the end of an array while processing data that directly or indirectly came from the user. In the general form, this attack provides a way for an attacker to change a program’s memory in a way not intended by the programmer, and allows the attacker to set variables or even change the code of the program

The specific buffer overrun attack used by the Morris worm was to send a line of input to the **`fingerd`** daemon. This daemon allowed remote users to request information about users on the local computer (for example, what was the user’s phone number). The daemon was constantly reading requests for service over the network using the **`gets`** function. The **`gets`** function does not check that the line of input fits within the destination array: if the line is too long, it stores to memory locations following the end of the array. The Morris worm took advantage of this by sending a large line input that when it overflowed the buffer wrote new code into the daemon. The worm then had control of a program with root privileges on the remote host.

Buffer overrun attacks continue to be a security problem. Roughly 10% of vulnerability reports cataloged by CERT from 01/01/2005 to 07/01/2005 involved buffer overflows. Preventing buffer overruns is the primary, but not the only, motivation for this technical report.

1.1 Goals

The committee had many goals in mind as it developed this technical report. In some cases, different goals conflict, and required the committee to make trade-offs. For example, the goal to have a uniform pattern for the function parameters and return type conflicts with the goal to minimize source code changes. In order to get the best result, the goals were evaluated for each function in this technical report individually. Thus, different functions balanced conflicting goals differently.

The remainder of the subclause lists the goals for the technical report. More important goals tend to be listed first, but this ordering is intrinsically loose because different goals had different importance for different functions.

1.1.1 Mitigate certain security vulnerabilities

Security is a big topic, and in its broadest sense, affects not just coding, but design philosophy, design technique, development approach, testing, deployment, system policy, and even use of programs. For example, the best designed, most carefully written program can be made insecure if installed with protections that allow its file to be modified. Likewise, security can be undone if a user protects a resource with an easily guessed password.

This technical report limits itself to a narrow aspect of security: functions to mitigate certain security problems. Those problems are:

1. Buffer overrun attacks
2. Attacks based on the **`%n printf`** specifier
3. Default protections associated with program-created files

(Buffer overrun attacks are discussed above. The other problems will be discussed with the functions affected.)

Note that this library only mitigates, that is lessens, these problems. When used properly, these functions decrease the danger from certain attacks, but any other security vulnerabilities remain. Programs can also still remain vulnerable due to bugs in the program (was the correct array size passed?) or even the implementation or the hardware. Security is always a matter of degree.

1.1.2 Guard against overflowing a buffer

This goal follows directly from the above goal. The functions in this technical report should not store data outside of its intended target. Whenever data is stored to an array, a bounds should be used verify that other storage is not being modified.

1.1.3 Do not produce unterminated strings

If a string lacks the terminating null character, the program may be tricked into accessing storage after the string as legitimate data. This may cause a program to process a string that it should not, which might be a security flaw in itself. It may also cause the program to abort, which might be a denial of service attack.

Note also the emphasis is not to produce unterminated strings. This library does not address processing of already existing unterminated strings (although the **`strnlen_s`** and **`wcsnlen_s`** function provide limited support for that). There are two reasons. First, if you prevent the creation of unterminated strings, then need to process them is greatly lessened. Second, if you associate a bounds with every string that is only used as input to a function, you vastly increase the size of the library, and require a much larger migration effort by existing programs, for comparatively little benefit.

1.1.4 Do not unexpectedly truncate strings

In general, when a function produces a string result, it should not quietly truncate the result to fit the output array. Such truncation may be a sign that an output buffer really should be larger. It might also be a security flaw: if a large string is verified for some purpose, and then unintentionally shortened, the new shorter string might not be valid for the same purpose. For example, if a daemon guards access to a group of files, it might verify that a particular 1000 character filepath was a valid access. If an erroneously small buffer then causes that filepath to be truncated to only 256 characters after the longer string was verified, the daemon would access the wrong file thinking that the path had been vetted.

1.1.5 Provide a library useful to existing code

The target client for this library is existing C code. The library should require only a reasonable effort to switch to the more secure functions in order to make feasible an improvement in security. If the costs are too high, existing code might never be modified, and thus not see any improvement in security.

The committee was fortunate to get feedback from organizations attempting to switch to this library as this technical report was being developed.

1.1.6 Preserve the null terminated string datatype

The null terminated string is a pervasive datatype, and is used by non-standard libraries as well as user-written code. A new datatype, such as a string representation with a built-in bounds, would have some advantages. For example, it would remove the requirement that the programmer keep track of the array size for strings separately. However, such a new datatype would not be limited to calls to the library functions in this technical report. In general, much of the user-written code that calls these library functions would have to be changed to also process the new datatype. A user function that calls the library with the new datatype would probably need to be modified to have parameters of the new datatype. Thus the caller of the user function would also have to change, which would in general require further changes to the program. In order to minimize that cost to migrate a program to the new library, the null terminated string datatype is the focus of this technical report.

1.1.7 Do not require size arguments for unmodified strings

Although functions that create strings have an additional parameter giving the number of elements in the target array, functions that do not modify strings are unchanged. This minimizes the number of new functions in this technical report, and allows edits to existing code to be minimal.

1.1.8 Only require local edits to programs

The functions in this library can replace their less secure counterparts with only a local change affecting only a line or two of code. These edits are almost mechanical in nature.

Limiting the scope of the edits makes it more economical and feasible to migrate a large program to this library.

See the **`gets_s`** function (Subclause 6.5.4.1) for a detailed discussion of this goal.

1.1.9 Library based solution

The committee wanted the technical report to concentrate on a library based solution. (The only requirement on compilers is predefined macro names, and in most implementations, they could be provided by compile switches or options without modifying the compiler itself.) A library based solution is easier to implement and distribute, and shortens the timeframe before security improvements could occur.

1.1.10 Support compile-time checking

While the committee wanted a solution that did not require compiler support, the committee was mindful that a compiler could be a useful tool in migrating a program to the new library. A compiler can flag calls to functions that should be replaced with calls to functions in this library. A compiler could have built-in knowledge of the **`scanf_s`** functions, and aid in getting their arguments right. A compiler could enforce checking return codes for functions returning **`errno_t`**. Footnotes were used in the technical report to point out situations where compiler support would be useful.

1.1.11 Make failures obvious

Both program correctness and security are harmed when a failure goes unnoticed and unhandled. Because of this, the library tries to make failures more obvious, so that it is unlikely that a program will quietly ignore the failure.

1.1.11.1 Zero buffers, null strings

One way failure is made more obvious is to produce a result that is obviously wrong. For example, the memory copy functions zero the output buffer if an error occurs. Likewise, the string functions produce a null string result if an error (such as unexpected truncation) occurs.

1.1.11.2 Runtime-constraint handler mechanism

If a library function detects an error, such as invalid arguments or not enough room in an output buffer, a special “runtime-constraint” handler function is called. This function might print a message and/or abort the program. The programmer has control of the handler function called via the **`set_constraint_handler_s`** function, and can make the handler simply return if desired. If the handler simply returns, the function that invoked the handler indicates a failure to its caller using its return value.

1.1.12 Support re-entrant code

The functions in the technical report should not rely on static internal state. Static internal state prevents the functions from being re-entrant, and leads to bugs.

1.1.13 Consistent naming scheme

Names of functions in the technical report end in "`_s`". This naming pattern makes clear that these functions are an extension to the standard library, that they came from this technical report, and also decreases the conflict with function names from other standards.

1.1.14 Have a uniform pattern for the function parameters and return type

Many functions in the technical report return a value of type **`errno_t`**. This typedef is used to indicate that the function is returning an error code normally associated with **`errno`**. In some cases, the committee thought that it would complicate migration to the library if a more secure function returned a different value than its less secure counterpart, and did not follow this pattern.

Functions have a parameter giving the number of elements in any array the function modifies. That parameter appears right after the parameter pointing to the array. The parameter's name usually includes the word "max." For example,

```
errno_t tmpnam_s(char *s, rsize_t maxsize);
```

1.1.15 Deference to existing technology

The committee has a long tradition of respecting existing technology, and prefers to standardize features and functions that have already proven themselves by actual use by programmers.

The committee considered three particularly important sources of existing technology while producing this technical report:

1. ISO/IEC 9945:2003, also known as the Single Unix Specification or POSIX
2. The OpenBSD functions strlcpy and strlcat
3. Experiences of companies performing security overhauls of large code bases

In most cases, the committee discovered that existing functions from the above sources would need modification to meet the goals of the technical report. For example, many existing functions lacked parameters giving the number of elements in an output array. Also, the runtime-constraint handler mechanism impacted every function (except **`strnlen_s`** and **`wcsnlen_s`**) in this technical report in two ways. First, the functions are required to invoke the handler when appropriate. Second, the function specifications now explicitly list conditions that would have been undefined behavior in ISO/IEC 9899:1999, and require specific behavior from the functions (such as returning error codes and leaving variables in known states) in addition to calling the handler.

The subclauses for different functions will discuss existing technology when appropriate.

2 References

The technical report references the expected related standards necessary to complete its specification. Chief among those is the C Standard itself, ISO/IEC 9899:1999, along with its Technical Corrigenda.

As Clause 1 of the technical report states, the technical report is to be read as if it was merged into C Standard. This has the effect of making statements and requirements in ISO/IEC 9899:1999 and its Technical Corrigenda apply to the technical report unless a corresponding section of this technical report states otherwise.

For example, Subclause 7.1.4 of ISO/IEC 9899:1999 permits any function declared in a standard header to be additionally implemented as a function-like macro. That permission extends to the functions in the technical report, since the sections of the technical report are to be read as if they were merged into ISO/IEC 9899:1999.

3 Terms, definitions, and symbols

The committee decided that many cases of what ISO/IEC 9899 would call *undefined behavior* should be detected and prevented when using the functions in the technical report. Examples of such undefined behavior include dereferencing a null pointer or storing a value to an array outside of the array bounds.

Such behaviors could no longer be called undefined behavior, since ISO/IEC 9899 permits the implementation to fail in an unpredictable way whenever undefined behavior occurs, while the technical report requires the implementation to behave in a specified and predictable way that potentially allows programs to recover from the problem (see Subclause 6.1.4).

The committee decided to call this new category of behavior a *runtime-constraint* based on its similarity to constraints in the Language Clause of ISO/IEC 9899. Constraints are violations of language rules that an implementation shall detect and diagnose. Runtime-constraints are violations of the runtime requirements of a function that the implementation must detect and diagnose by a call to a handler and, if the handler returns, by a failure indicator of some kind returned to the caller of the “failed” function call. Like constraints, runtime-constraints appear in special subheaders in the document as statements using the words “shall” or “shall not” to place requirements on the program.

Note that runtime-constraints are disjoint from constraints. Constraints are rules the program must follow during translation time. Runtime-constraints are rules that the program must follow at runtime. A runtime-constraint is not a special case of constraint: it is merely a parallel concept.

Other names considered for runtime-constraints were “diagnosed undefined behavior” and “usage requirements.”

4 Conformance

The Clause in ISO/IEC 9899 corresponding to this one needed to be modified to acknowledge that “shall” and “shall not” requirements also appear in runtime-constraints sections.

5 Predefined macro names

A macro is provided to allow users to determine if the technical report is supported by the implementation. The value of the macro is the year and month that the features of the technical report were last changed significantly.

Most implementations provide some way to predefine macros using command line or programming environment options. Thus, the compiler or preprocessor need not be modified to produce an implementation that conforms to the technical report. A library-only solution can simply require that programmers make use of the facilities to predefine the required macro.

6 Library

6.1 Introduction

6.1.1 Standard headers

ISO/IEC 9899 in Subclause 7.1.3 prohibits adding additional functions to the standard headers unless the names of the additional functions match certain patterns of reserved identifiers. The rationale for this prohibition is that adding additional identifiers to a standard header potentially breaks strictly conforming programs, and a conforming implementation must accept every strictly conforming program, subject to translation limits.

For example, the following is a strictly conforming program:

```
#include <stdio.h>
int myfunc(void) {return 0;}
int main(int argc, char **argv) {return myfunc();}
```

However, if the implementation added the following prototype to **`<stdio.h>`**:

```
extern float myfunc(char *);
```

then the program would contain a constraint violation since all the declarations of **`myfunc`** in the same scope would not have compatible types.

The technical report adds many identifiers to standard headers that do not match reserved identifiers. To prevent this from making an implementation not conform to ISO/IEC 9899, the functions, type names, and macros added by the technical report are under the control of a macro named **`__STDC_WANT_LIB_EXT1__`**, whose name does match the pattern of reserved names in ISO/IEC 9899. (Typographical note: the macro name begins and ends in two underscore characters.)

If this macro is defined to be 1, the additional identifiers in the technical report are defined by their respective headers. If the macro is defined to be 0, the additional identifiers are not defined, and the implementation is (not prevented from being) conforming to ISO/IEC 9899.

If the macro is not defined, the implementation may choose to behave as if the macro was defined to be either 1 or 0. Many implementations do not conform to ISO/IEC 9899 by default, and one of the most frequent reasons for that is the desire to define extra functions in standard headers, particularly functions required by other standards, such as POSIX. The committee decided to allow implementations to acknowledge this marketplace reality, and allow implementations to do what is best for their customers.

Note that most implementations provide a way to predefine a macro in the command line or build environment options. Programmers need not change their sources to define **`__STDC_WANT_LIB_EXT1__`**.

Some of the identifiers defined in the technical report do match reserved name patterns in ISO/IEC 9899, and thus do not raise conformance issues. However, the committee though it cleaner if all identifiers were uniformly protected by **`__STDC_WANT_LIB_EXT1__`** rather than only the ones that needed it. This also eliminates conflicts with implementations and programs that intruded into the reserved identifiers.

6.1.2 Reserved identifiers

This subclause duplicated from ISO/IEC 9899:1999 for the purposes of exposition.

6.1.3 Use of `errno`

`errno` has fallen into disfavor, and the committee largely considers it a traditional feature maintained for compatibility. The technical report allows an implementation to set **`errno`**, but does not require it to do so. In general, functions in the technical report return some sort of indication of failure, and make **`errno`** superfluous.

See Subclause 6.2.

6.1.4 Runtime-constraint violations

Except for **`strnlen_s`** and **`wcsnlen_s`**, functions in the technical report have a "Runtime-constraints" section that lists a series of "shall" or "shall not" statements that the program must satisfy when calling a library function. The implementation is required to enforce the runtime-constraints. Typically, this is done by the library function checking the conditions immediately upon entry, or as it is performing its task and gathers enough information to make a decision about a particular runtime-constraint.

Some “Runtime-constraints” sections contain prohibitions (e.g., the function does not modify the object pointed to by a parameter) or requirements (e.g., the function stores zero in the object pointed to by a parameter) that apply if any runtime-constraint is violated by a function. The function must not do anything prohibited, and must do anything required by the “Runtime-constraints” section before calling the handler. See Subclause 6.6.1.

Should the handler return, the function immediately returns a value to its caller. The “Returns” section of the function will describe the value returned if a runtime-constraint occurs and the handler returns.

The runtime-constraints of functions in the technical report are conditions that would be undefined behavior for a function in ISO/IEC 9899. This technical report eliminates much undefined behavior, but undefined behavior still exists. Some cases of undefined behavior are too expensive to detect for many implementations, and the functions defined in ISO/IEC 9899:1999 have whatever undefined behaviors specified in that standard.

However, ISO/IEC 9899:1999 defines undefined behavior as “behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.” Since there are no requirements, an implementation is free to turn any undefined behavior into a runtime-constraint violation. This is true of functions in ISO/IEC 9899:1999 as well as functions in this technical report.

6.2 Errors <errno.h>

Although **`errno`** itself is considered somewhat outmoded (see Subclause 6.1.3), the concept of a set of **`errno`** values to indicate various failure conditions is used by many functions in the technical report. Many functions return a value that would be the value to which the functions set **`errno`**, if the functions did set **`errno`**. Although ISO/IEC 9899:1999 defines only three different specific values for errno, other standards and conventions define many more.

Because of the usefulness that the set of errno values represents, the technical report defines a typedef, **`errno_t`**, to represent this set of values. The type of **`errno_t`** is required to be **`int`**, which is also the type of **`errno`** itself.

The fact that **`errno_t`** must be **`int`** sets it apart from typedefs in ISO/IEC 9899. ISO/IEC 9899 has comparatively few typedefs, and all of them represent types that may differ between implementations.

The committee recognized that **`errno`**_t is very useful pedagogically, and that declaring a function to return **`errno`**_t is a valuable shorthand to express the true high-level significance of the return value to the programmer. Declaring types as an aid to understanding is part of modern programming style. Such types are beneficial even if they do not vary between implementations.

6.3 Common definitions <stddef.h>

The type **`rsize_t`** is similar to **`size_t`** (see Subclause 6.4) and like **`size_t`**, is declared in the headers that use it. Since it is defined in multiple headers, the declarations need to be protected with conditional inclusion based on a macro whose name matches a reserved pattern from ISO/IEC 9899:1999 Subclause 7.1.3. For example:

```
#if __STDC_WANT_LIB_EXT1__ == 1
#ifndef __RSIZE_T
#define __RSIZE_T
typedef size_t rsize_t;
#endif
#endif
```

See Subclause 6.2 for a discussion on the pedagogical use of typedefs.

6.4 Integer types <stdint.h>

A common error when calculating the size of objects is to produce a “negative” number. There are several ways this might happen. Pointers might be subtracted in the wrong order. The size might be updated when bytes are used in the object, and a bug might cause too many bytes to be used. Mixed 64-bit and 32-bit code might erroneously sign extend a count that should not be extended. Although the “true” arithmetic value of a calculated size might be negative, the **`size_t`** type is unsigned, and a negative value stored in it will appear as a large positive value.

Anytime the size of an object is wrong, there is the possibility that bytes outside of the object might be modified when storing to the object. In addition to being undefined behavior, this might a vulnerability that could be exploited by a buffer overrun attack.

The committee wished to allow implementations to place reasonable limits on the size of objects, so that suspiciously large object sizes could be runtime-constraint violations. The typedef **`rsize_t`** is has the same type as **`size_t`**, but indicates that functions that have parameters of type **`rsize_t`** will range check the value of the parameter. The macro **`RSIZE_MAX`** is the upper limit for runtime-constraint checking of the values of type **`rsize_t`**.

An implementation is free to make **`RSIZE_MAX`** the maximum value that the representation of the type **`rsize_t`** can store. In that case, all values of type **`rsize_t`** are not runtime-constraint violations, and the functions in the technical report need not check the values of any **`rsize_t`** parameters.

The **`RSIZE_MAX`** macro need not expand to a constant expression. Some implementations might wish to adjust the value of **`RSIZE_MAX`** dynamically. For example, **`RSIZE_MAX`** might reflect the amount of memory actually allocated to the program as opposed to the size of the address space. Another example, implementations that support both 32-bit and 64-bit address spaces might determine the appropriate value for **`RSIZE_MAX`** at runtime. The committee rejected a suggestion to provide a function to change the value of **`RSIZE_MAX`** at runtime: it is premature to adopt such an interface until its need has been proven.

Note that **`RSIZE_MAX`** is the limit on the size of any single object: it is not the limit of all objects taken together. For example, if **`RSIZE_MAX`** was a fraction of the size of the address space, multiple objects might be allocated to completely fill the address space. Thus, **`RSIZE_MAX`** does not limit the total amount of memory a program might allocate.

6.5 Input/output <stdio.h>

6.5.1 Operations on files

The two functions in this subclause, **`tmpfile_s`** and **`tmpnam_s`**, deal with implementation-generated temporary files. Some implementations chose the patterns used to name temporary files years ago when filesystems placed draconian limits on length of files and before multithreaded applications were common. Because of this, on those implementations, the existing C functions generate temporary file names that are too short and too prone to race conditions. The race conditions might not only occur in multithreaded applications, but even when the same user runs multiple copies of the same application.

Unfortunately, changing the algorithm that generates temporary filenames, especially if the length of the temporary filenames grows, can be compatibility problem. For example, the **`tmpnam`** function in ISO/IEC 9899:1999 assumes that the array to be used to store the result is big enough (the number of elements in the array is not passed to **`tmpnam`**). The macro **`L_tmpnam`** can be used to declare arrays of the proper size for use with **`tmpnam`**. However, **`L_tmpnam`** is a macro whose value is an integer constant expression fixed at compile-time. If the library is dynamically linked to the application rather than statically linked, a new **`tmpnam`** that requires larger arrays would overwrite the end of the result array in any program not recompiled.

By providing two new functions, **`tmpfile_s`** and **`tmpnam_s`**, implementations get a chance to modernize their algorithm for generating temporary file names.

6.5.1.1 The tmpfile_s function

In addition to the improvements given in Subclause 6.5.1, the **`tmpfile_s`** function protects the temporary file from unauthorized access by setting its file protection and opening the file with exclusive access.

6.5.1.2 The tmpnam_s function

In addition to the improvements given in Subclause 6.5.1, the **`tmpnam_s`** function is protected from overwriting the end of the result array.

Programs that use the **tmpnam_s** function have a potential race condition. After the program obtains a temporary name using **tmpnam_s**, but before the program can create a file using that name, someone else may create a file with that same name. The possibility of this problem can be reduced, but not eliminated, if the implementation chooses temporary filenames that are long, unusual, and contain a thread id.
Because of this race condition, **tmpfile_s** should be used when possible. But, if the program needs to repeatedly open and close the temporary file, or to create a temporary directory rather than a file, **tmpnam_s** should be used.

6.5.2 File access functions

When creating a file, the **fopen_s** and **freopen**_s functions improve security by protecting the file from unauthorized access by setting its file protection and opening the file with exclusive access.

6.5.3 Formatted input/output functions

6.5.3.1 The printf family of functions

The printf family of functions is susceptible to a variety of security attacks if the format string comes directly or indirectly from the user. Consider a program like the following:

```
#include <stdio.h>
int main(int argc, char **argv)
{
    printf(argv);
    return 0;
}
```

If a user runs this program with the argument string "**%x,%x,%x**", the program might print various stack entries (this is undefined behavior). If one of the values appears to be an address, running the program again with a properly placed **%s** in the argument string might cause storage at that address to be displayed. Carefully crafted, potentially very long format strings could be prepared by an attacker to view an arbitrary number of bytes back in the stack. Since the format itself may be on the stack, the attacker might use these techniques to search for the format, and then construct a format that references its own locations as the arguments corresponding to its conversion specifiers. Since the format can contain literal characters in addition to conversion specifiers, the format might reference these literal characters as the argument corresponding to a **%s** specifier in order to dump memory from any address the attacker chooses.

In a simple program, the dangers of such probing are minimal. However, some real world programs have variables that contain confidential information that is not normally displayed by the program. Techniques like the above could cause the program to divulge such information to an unauthorized user.

Even worse, the **`%n`** format specifier can be used to change memory in the program. The **`%n`** specifier stores the number of characters written thus far by the printf function to the value pointed to by the corresponding argument. Since the corresponding argument might be literal characters in the format itself, an arbitrary address may be the target of the **`%n`** specifier. By performing overlapping stores that offset from each other by a single byte, an arbitrary sequence of bytes could be stored.

The committee briefly discussed requiring formats to be compile-time constants. However, internationalization frequently requires that all formats come from an external source, such as a message catalog.

The committee decided to merely warn in this Rationale of the dangers of formats from untrusted sources, and to remove support for the **`%n`** specifier in a new **`_s`** family of **`printf`** functions.

Format attacks are very difficult, but they have been made against real programs. The wu-ftpd FTP daemon contained a format string vulnerability that allowed remote users to gain root privileges on UNIX and LINUX systems.

Implementations might wish to implement an optional warning for non-literal format strings.

Some implementations know at runtime the actual number of arguments passed to functions that take a variable number of arguments. Such implementations might use such information to recognize a runtime-constraint violation if a conversion specifier does not correspond to an argument actually passed in the function call.

Implementations should review any extensions they have made to format specifiers. If any extension allows the format string to modify memory like **`%n`**, that extension should not be supported in the new **`_s`** family of **`printf`** functions.
.

6.5.3.1.1 snprintf/vsnprintf versus sprintf/vsprintf

The **`snprintf`** function was an earlier attempt by the committee to fix buffer overflows associated with **`sprintf`**. Once **`%n`** conversion specifiers were prohibited in both **`snprintf_s`** and **`sprintf_s`**, and once the buffer overflow was fixed in **`sprintf_s`**, the committee was left with two fairly similar functions.

The committee decided to use one of the functions to support truncation (Subclause 1.1.4). The **`snprintf_s`** function will truncate its result to fit the supplied output array, while **`sprintf_s`** will raise a runtime-constraint violation if the result does not fit. The justification for supporting truncation in **`snprintf_s`** is that **`snprintf`** supported truncation, and the printf functionality is so complex it is hard for a programmer to force truncation when it is needed unless it is explicitly supported by a **`printf`**-family function.

The **`sprintf_s`** function differs from other **`printf`**-family functions (including **`sprintf`** itself) in its return value. Unlike the other functions, it returns zero rather than a negative number if a runtime-constraint violation occurs. The reason for this if the committee wanted to allow the return value of **`sprintf_s`** to be added to a running total that keeps track of the number of characters written to a string. The **`sprintf`** function could be used that way on some implementations. The **`sprintf`** function only returns a negative value if a multibyte encoding error occurs, and some implementations do not have any encoding errors. Programmers on such systems might rely on that property, and so **`sprintf_s`** only returns negative values for the same cases as **`sprintf`**.

6.5.3.2 The scanf family of functions

The **`scanf`** family of functions is even more vulnerable than the **`printf`** family if the format comes from an untrusted source (see Subclause 6.5.3.1) since all the specifiers in **`scanf`**-family functions allow memory to be modified. Do not obtain a **`scanf`**-family format directly or indirectly from the user. Implementations might wish to implement an optional warning for non-literal format strings.

The **`scanf`**-family functions also contain a much easier to exploit vulnerability. They do not check the size of arrays used to store the results of **`%c`**, **`%s`**, and **`%[`** conversion specifiers. (Some people might think the **`%c`** specifier does not write to an array since without a field width, it reads a single character. However, this is merely the degenerate case of an array of length one. The **`%c`** specifier reads an array of however many characters specified by its field width, and by default the field width is 1.)

The new **`_s`** family of **`scanf`** functions fixes the buffer overrun vulnerability by requiring an extra argument after any argument corresponding to any **`%c`**, **`%s`**, and **`%[`** conversion specifiers. (Even %c specifiers without a field width take the extra argument.) The extra argument gives the number of elements in the array that is the target of the specifier.

The committee considered an alternative proposal that would require a precision of "**`.*`**" be used with any **`%c`**, **`%s`**, and **`%[`** conversion specifier in the format of one of the new **`_s`** family of **`scanf`** functions. This would have the same effect as the solution that was accepted: an argument corresponding to one of these specifiers would be immediately followed by another argument giving its length. However, this approach would force programmers to modify not only the argument list when migrating to the new **`_s`** **`scanf`** functions, but the format string as well. Since the format string might not be a literal, and might not even be present in the same source file as the function call, this was seen as too great a burden.

Some implementations know at runtime the actual number of arguments passed to functions that take a variable number of arguments. Such implementations might use such information to recognize a runtime-constraint violation if a conversion specifier does not correspond to an argument actually passed in the function call.

6.5.4 Character input/output functions

6.5.4.1 The gets_s function

The **`gets_s`** function might be the best example of a function added to meet the goal of only requiring only small, local edits (Subclause 1.1.8) to migrate to the new library.

The **`gets`** function was exploited by the original buffer overrun attack used by the Great Internet Worm of 1988. The 1989 C Standard contained a better version of this function, **`fgets`**, that fixed the vulnerability. Yet, some programs still use **`gets`**. The reason for this is the differences between **`fgets`** and **`gets`** that make **`fgets`** a better function also complicate switching from **`gets`** to **`fgets`**.

The programmer who uses **`gets`** has made two (perhaps unwise) simplifying assumptions that affect the way the code is written. The first is that every call to **`gets`** will read one line of input, or will encounter EOF (or an I/O error). The second is that every line of input will fit in the array supplied as an argument to gets. Furthermore, the line returned by **`gets`** will have any newline character stripped from the string.

In contrast, **`fgets`** does not always read a full line of input every time it is called: it only reads as much as will fit in the supplied array. Furthermore, the returned string may or may not be terminated with a newline. The lack of a newline indicates that either the input line was longer that the array to store it (and more calls to fgets are needed to finish reading the line) or that the line is an unterminated line before EOF or an input error. A correct program that uses **`fgets`** either needs to be able to process partial lines or to loop to finish reading the full line of input before processing it. If the program also depends upon the newline character being deleted from the input line, the program will have to delete the newline itself.

The **`gets`** function is a security vulnerability, and programs should not use it. However, **`fgets`**, while having many advantages, may also require too much effort as an alternative. (Even a modest increase in effort may derail an effort to modernize a program that is tens or hundreds of thousands of lines long.)

The committee decided to provide a function, **`gets_s`**, which allows programs to keep the assumptions they have when using **`gets`**, but the function makes it a runtime-constraint violation if those assumptions are violated. Every successful call to **`gets`**_s reads a full line of input (and deletes the newline). Every successful call fits the input line into the supplied array. If the call to **`gets_s`** is not successful, for example because the array is not big enough to hold the full line of input, the runtime-constraint handler is called, and if the handler returns, **`gets_s`** returns a null pointer to indicate the failure of the input operation.

Programmers can use **`gets_s`** to fix the **`gets`** security vulnerability in existing programs without having to think through the issues involved in migrating to **`fgets`**. A project fixing a large code base need not take the position, we will fix the easy functions now, and deal with **`gets`** when we have time.

There is another case where **`gets_s`** might prove useful. Some earlier projects to fix security problems in code bases might have blindly replaced **`gets`** calls with **`fgets`** calls without thinking through the issues described above. Such programs might be better served by changing now problematic **`fgets`** calls to **`gets_s`** calls.

6.6 General utilities <stdlib.h>

6.6.1 Runtime-constraint handling

The **`set_constraint_handler_s`** function allows the programmer to control the handler that is called by functions in the technical report when a runtime-constraint violation occurs (Subclause 6.1.4). The runtime-constraint handler mechanism makes violations more visible (Subclause 1.1.11).

The second argument to the handler allows an implementation to pass additional information to the handler. For example, the implementation might pass a pointer to an object giving the name of the function that detected the runtime-constraint violation and the line number when the violation was detected.

The **`abort_handler_s`** and **`ignore_handler_s`** functions represent handlers for two common situations, and are provided simply for convenience. Note that the implementation default handler need not be either **`abort_handler_s`** or **`ignore_handler_s`**. (The implementation default handler is used if **`set_constraint_handler_s`** has never been called or if a null pointer is passed as the **`handler`** argument to **`set_constraint_handler_s`**.)

Programmers may wish to implement runtime-constraint checking in their own code, and to call the current runtime-constraint handler when a violation is detected. The following code fragment gets a pointer to the currently registered runtime-constraint handler and then calls it:

```
// Get the current handler
constraint_handler_t handler =
        set_constraint_handler_s(NULL);
// Restore the current handler
set_constraint_handler_s(handler);
// Call it to report a domain error
handler("Domain Error", NULL, EDOM);
```

Most implementations will probably use a pointer to function in their implementation of the **`set_constraint_handler_s`** function to hold the address of the currently registered handler. Unfortunately, pointers to functions are employed by many security exploits. If an exploit deposits new code into a program, the exploit must find a way to cause that new code to be executed. The most common way is to alter the return address on the stack to point to the new code, but an alternative is to find a pointer to a function, and store the address of the new code in that pointer.

The committee thought that the benefit of a user-settable runtime-constraint handler justified providing another pointer to function that might be exploited. There are steps that an implementation can take to mitigate the vulnerability of the pointer. Some possibilities are:

- Dynamically allocate the pointer or otherwise arrange for the address of the pointer to change every time the program runs
- Write-protect the page containing the pointer, and have **`set_constraint_handler_s`** only write-enable the storage when it is updating the pointer's value.
- "Encode" the value of the pointer, so that it is not a pure address.

Note that anytime a program terminates by a call to the abort function, including when a runtime-constraint handler calls abort, that some resources managed by the program might not be released. For example, output buffers may not be flushed and temporary files may not be deleted.

6.6.2 Communication with the environment

6.6.2.1 The getenv_s function

The **`getenv_s`** fixes the reentrancy problems with gets (Subclause 1.1.12) and fixes a possible buffer overflow. The **`getenv_s`** function can also be used to get the size needed to represent the result. This allows the programmer to first call **`getenv_s`** to get the size, then allocate a buffer to hold the result, and then call **`getenv_s`** again to actually obtain the result.

6.6.3 Searching and sorting utilities

The **`bsearch_s`** and **`qsort_s`** functions allow a context argument to be passed to the comparison function. This allows for more sophisticated comparisons. For example, the comparison might be done in a specific locale or with a private collation table. Without the extra argument, either the programmer would have to write separate comparison functions for each "context," or would have to use global variables to provide the extra "context" to the comparison function. The **`bsearch`**_s and **`qsort`**_s functions remove this reentrancy problem (Subclause 1.1.12).

An early implementation of the **`bsearch_s`** and **`qsort_s`** functions that performed runtime-constraint-like checks discovered that many legitimate uses of these functions operated on arrays of zero elements. The committee decided to require reasonable behavior if the number of elements in the array was zero (**`bsearch_s`** fails to find the key and **`qsort_s`** does not alter the array).

6.6.4 Multibyte/wide character conversion functions

The **`wctomb_s`** function adds an extra parameter to prevent a buffer overflow.

The **`wctomb_s`** function is designed to be used in loops that process a string a character at a time. As such, it is not appropriate for **`wctomb_s`** to null terminate its result.

The **`wctomb_s`** function has internal state, which is a reentrancy problem (Subclause 1.1.12). The **`wcrtomb_s`** function (Subclause 6.9.3) fixes this problem, and should be used when possible. The **`wctomb_s`** function is provided because requiring the program to manage the conversion state may complicate migrating to the more secure functions in the technical report (Subclause 1.1.8).

6.6.5 Multibyte/wide string conversion functions

The **`mbstowcs_s`** and **`wcstombs_s`** functions have an additional parameter giving the size of the array that is the destination of the conversion in order to prevent buffer overflow.

These functions have a feature lacking in the **`mbstowcs`** and **`wcstombs`** functions: If the destination pointer is null, **`mbstowcs_s`** and **`wcstombs_s`** will store the length of the result. This allows a program to call these functions to determine the amount of space needed, then to allocate space for the result, and then call these functions a second time to actually obtain the result.

The **`mbstowcs_s`** and **`wcstombs_s`** functions have internal state, which is a reentrancy problem (Subclause 1.1.12). The **`mbsrtowcs_s`** and **`wcsrtombs_s`** functions (Subclause 6.9.3) fix this problem, and should be used when possible. The **`mbstowcs_s`** and **`wcstombs_s`** functions are provided because requiring the program to manage the conversion state may complicate migrating to the more secure functions in the technical report (Subclause 1.1.8).

The **`mbstowcs_s`** and **`wcstombs_s`** functions can be implemented as a wrapper around **`mbsrtowcs_s`** and **`wcsrtombs_s`**.

6.7 String handling <string.h>

6.7.1 Copying functions

6.7.1.1 The memcpy_s function

The **`memcpy_s`** function has an additional parameter giving the size of the destination array in order to prevent buffer overflow. If a runtime-constraint violation occurs, the destination array is zeroed to increase the visibility of the problem (Subclause 1.1.11).

In order to reduce number of cases of undefined behavior, the **`memcpy_s`** function must report a constraint-violation if an attempt is being made to copy overlapping objects. For some functions in the library (for example, the **`printf_s`** and **`scanf_s`** functions), detecting overlapping operands is too difficult to be practical. However, experience with the **`memmove`** function has shown that it is practical to detect overlapping operands in a **`memcpy`**-like function.

6.7.1.2 The memmove_s function

The **`memmove_s`** function has an additional parameter giving the size of the destination array in order to prevent buffer overflow. If a runtime-constraint violation occurs, the destination array is zeroed to increase the visibility of the problem (Subclause 1.1.11).

6.7.1.3 The strcpy_s function

The **`strcpy_s`** function has an additional parameter giving the size of the destination array in order to prevent buffer overflow. If a runtime-constraint violation occurs, the destination array is set to a null string to increase the visibility of the problem (Subclause 1.1.11).

Because truncating a source string to fit in the destination can be a security vulnerability (Subclause 1.1.4), the **`strcpy_s`** function does not truncate, and treats such cases as a runtime-constraint violation. However, if the programmer wishes to force truncation, there is an idiom using **`strncpy_s`** (See Subclause 6.7.1.4) that can be used.

The **`strcpy_s`** function is similar to the OpenBSD function **`strlcpy`**, but has some important differences. The **`strlcpy`** function truncates the source string to fit in the destination if the destination is shorter than the source. Since truncation is a possible security vulnerability, the committee decided this was unacceptable. The **`strlcpy`** function does not perform all of the runtime-constraint checks that **`strcpy_s`** does, and so is less robust. The **`strlcpy`** function does not make failures obvious by setting the destination to a null string or calling a handler if the call fails. The **`strlcpy`** function has been criticized by some programmers as forcing them to check its return value to see if the function failed. The committee decided to give such programmers the option to delegate that job to an automatically called handler when using **`strcpy_s`**.

6.7.1.3.1 Overlapping operands

The **`strcpy_s`** function must detect a runtime-constraint violation if its source and destination operands overlap. Unlike the **`memcpy_s`** function, this is harder to detect since the length of its source operand is not immediately available. While it would be possible to call **`strnlen_s`** to get the source string's length, this would cause **`strcpy_s`** to become a two pass algorithm. The first pass would walk the source string to get its length. The second pass would walk the source string to copy it. This would likely make the function twice as slow.

A more efficient way to check the overlap runtime-constraint is to attempt to copy the source to the destination. A side-effect of copying the source is determining its length.

Then, the overlap runtime-constraint can be easily checked, and the function need only walk the source string once.

This single pass algorithm is permitted for two reasons. First, runtime-constraints need not be checked when the function is first entered. The only requirement is that the runtime-constraints be checked early enough to prevent the function from performing any action that is prohibited if a runtime-constraint violation occurs. There is nothing prohibited about copying the source or modifying the destination in the specification of the **`strcpy_s`** function. Second, the specification of the **`strcpy_s`** function permits the function to modify every byte in the destination after the terminating null character. If a runtime-constraint violation occurs, a null character is written to the first element of the destination. Thus, if a runtime-constraint occurs, the function was permitted to attempt the copy and smash all of the elements of the destination array.

6.7.1.4 The strncpy_s function

The **`strncpy_s`** function has an additional parameter giving the size of the destination array in order to prevent buffer overflow. If a runtime-constraint violation occurs, the destination array is set to a null string to increase the visibility of the problem (Subclause 1.1.11).

The **`strncpy_s`** function stops copying the source string to the destination array when the first of the following two conditions occurs:

1. The null terminating the source string is copied to the destination.
2. The number of characters specified by the **`n`** parameter have been copied

The result in the destination is provided with a null character terminator if one was not copied from the source. The result including the null terminator must fit within the destination or a runtime-constraint violation occurs. Storage outside of the destination array is never modified.

Because the number of characters in the source is limited by the **`n`** parameter and the destination has a separate parameter giving the maximum number of elements in the destination, the **`strncpy_s`** function can copy a substring safely, not just an entire string or its tail.

Because unexpected string truncation is a possible security vulnerability (Subclause 1.1.4), **`strncpy_s`** does not truncate the source (as delimited by the null terminator and the **`n`** parameter) to fit the destination. Truncation is a runtime-constraint violation. However, there is an idiom which allows a program to force truncation using the **`strncpy_s`** function. If the **`n`** argument is the number of elements minus one in the destination, **`strncpy_s`** will copy the entire source to the destination or truncate it to fit (as always, the result will be null terminated). For example, the following call will copy **`src`** to the **`dest`** array resulting in a properly null terminated string in **`dest`**. The copy will stop when **`dest`** is full (including the null terminator) or when all of **`src`** has been copied:

```
    strncpy_s(dest, sizeof dest, src, (sizeof dest)-1);
```

While OpenBSD function **`strlcpy`** is similar to **`strncpy`**, it is more similar to **`strcpy_s`** than **`strncpy_s`**. Unlike **`strlcpy`**, **`strncpy_s`** does support copying substrings in a safe and secure manner. For more discussion of **`strlcpy`**, see Subclause 6.7.1.3.

The issues with detecting the overlapping operands runtime-constraint are similar to those in Subclause 6.7.1.3.1.

6.7.2 Concatenation functions

6.7.2.1 The strcat_s function

The **`strcat_s`** function has an additional parameter, **`s1max`**, giving the size of the destination array in order to prevent buffer overflow. The original string in the destination plus the new characters appended from the source must fit and be null terminated to avoid a runtime-constraint violation. If a runtime-constraint violation occurs, the destination array is set to a null string to increase the visibility of the problem (Subclause 1.1.11).

Because truncating a source string to fit in the destination can be a security vulnerability (Subclause 1.1.4), the **`strcat_s`** function does not truncate, and treats such cases as a runtime-constraint violation. However, if the programmer wishes to force truncation, there is an idiom using **`strncat_s`** (See Subclause 6.7.2.2) that can be used.

The **`strcat_s`** function is similar to the OpenBSD function **`strlcat`**, but has some important differences. The **`strlcat`** function truncates the source string to fit in the destination if the free space in the destination is shorter than the source. Since truncation is a possible security vulnerability, the committee decided this was unacceptable. The **`strlcat`** function does not perform all of the runtime-constraint checks that **`strcat_s`** does, and so is less robust. The **`strlcat`** function does not make failures obvious by setting the destination to a null string or calling a handler if the call fails. The **`strlcat`** function has been criticized by some programmers as forcing them to check its return value to see if the function failed. The committee decided to give such programmers the option to delegate that job to an automatically called handler when using **`strcat_s`**.

The issues with detecting the overlapping operands runtime-constraint are similar to those in Subclause 6.7.1.3.1.

6.7.2.2 The strncat_s function

The **`strncat_s`** function has an additional parameter giving the size of the destination array in order to prevent buffer overflow. The original string in the destination plus the new characters appended from the source must fit and be null terminated to avoid a runtime-constraint violation. If a runtime-constraint violation occurs, the destination array is set to a null string to increase the visibility of the problem (Subclause 1.1.11).

The **`strncat_s`** function stops appending the source string to the destination array when the first of the following two conditions occurs:

1. The null terminating the source string is copied to the destination.
2. The number of characters specified by the **`n`** parameter have been copied

The result in the destination is provided with a null character terminator if one was not copied from the source. The result including the null terminator must fit within the destination or a runtime-constraint violation occurs. Storage outside of the destination array is never modified.

Because the number of characters in the source is limited by the **`n`** parameter and the destination has a separate parameter giving the maximum number of elements in the destination, the **`strncat_s`** function can append a substring safely, not just an entire string or its tail.

Because unexpected string truncation is a possible security vulnerability (Subclause 1.1.4), **`strncat_s`** does not truncate the source (as specified by the null terminator and the **`n`** parameter) to fit the destination. Truncation is a runtime-constraint violation. However, there is an idiom which allows a program to force truncation using the **`strncat_s`** function. If the **`n`** argument is the number of elements minus one remaining in the destination, **`strncat_s`** will append the entire source to the destination or truncate it to fit (as always, the result will be null terminated). For example, the following call will append **`src`** to the **`dest`** array resulting in a properly null terminated string in **`dest`**. The concatenation will stop when **`dest`** is full (including the null terminator) or when all of **`src`** has been appended:

```
    strncat_s(dest, sizeof dest, src,
              (sizeof dest)-strnlen_s(dest, sizeof dest)-1);
```

While OpenBSD function **`strlcat`** is similar to **`strncat`**, it is more similar to **`strcat_s`** than **`strncat_s`**. Unlike **`strlcat`**, **`strncat_s`** does support appending substrings in a safe and secure manner. For more discussion of **`strlcat`**, see Subclause 6.7.2.1.

The issues with detecting the overlapping operands runtime-constraint are similar to those in Subclause 6.7.1.3.1.

6.7.3 Search functions

6.7.3.1 The strtok_s function

The **`strtok_s`** function fixes two problems in the **`strtok`** function:

1. A new parameter, **`s1max`**, prevents **`strtok_s`** from storing outside of the string being tokenized. (The string being divided into tokens is both an input and output of the function since **`strtok_s`** stores null characters into the string.)
2. A new parameter, **`ptr`**, eliminates the static internal state that prevents **`strtok`** from being re-entrant (Subclause 1.1.12). (The ISO/IEC 9899 function **`wcstok`** and the ISO/IEC 9945 (POSIX) function **`strtok_r`** fix this problem identically.)

The **`strtok_s`** function differs from the POSIX **`strtok_r`** function by guarding against storing outside of the string being tokenized, and by checking runtime-constraints.

Some might point out that the **`strtok`** function has a somewhat clumsy interface, and question whether it is wise to provide a function with similar shortcomings. For example, **`strtok`** modifies the string that it tokenizes, which make it unsuitable to parse a **`const`** string or a string literal. However, providing a more secure version of strtok is consistent with the goals of the technical report (Subclauses 1.1.2, 1.1.5, and 1.1.8).

6.7.4 Miscellaneous functions

6.7.4.1 The strerror_s function

The **`strerror_s`** function has an additional parameter (compared to **`strerror`**) giving the size of the destination array in order to prevent buffer overflow.

Unlike the other functions in the technical report, the **`strerror_s`** function supports string truncation. If the error message is too long for the destination, it is truncated to fit. A terminating ellipsis is added to the result to indicate that truncation occurred. The result is always a properly null terminated string that fits within the destination array.

The justification for supporting truncation in this function is that its purpose is to obtain an error message when something goes wrong. The last thing many programs will do before aborting is to display an error message obtained by **`strerror_s`**. Given this use, providing as much information about what went wrong is desirable.

The use of an ellipsis to indicate that the message string was truncated is consistent with other uses of ellipsis in C programming (for example, function prototypes), and in this case is more of a C Language cultural convention than an English language one.

6.7.4.2 The strerrorlen_s function

The committee received several requests for a new function to obtain the full, untruncated length of the message string that **`strerror_s`** would return. This would allow a program to determine the size of the array needed to store a result from **`strerror_s`** so that the program could allocate the buffer before calling **`strerror_s`**.

The Open Group and others specifically requested that a new function be used for this purpose, rather than having **`strerror_s`** return the length of the full message. Thus, **`strerrorlen_s`** was added to the technical report.

6.7.4.3 The strnlen_s function

The **`strnlen_s`** function is useful when dealing with strings that might lack their terminating null character. That the function returns the number of elements in the array when no terminating null character is found causes many calculations to be more straightforward. The technical report itself uses **`strnlen_s`** extensively in expressing the runtime-constraints of functions.

The **`strnlen_s`** function is identical the Linux function **`strnlen`**.

Because functions in the technical report do not produce unterminated strings (Subclause 1.1.3), in most cases there is no need to replace calls to the **`strlen`** function with calls to **`strnlen_s`**.

6.8 Date and time <time.h>

6.8.1 Components of time

The concept of a *normalized* time existed in ISO/IEC 9899, but was never named. For convenience, the term is defined here in the technical report.

6.8.2 Time conversion functions

6.8.2.1 The asctime_s function

The **`asctime_s`** function fixes static internal state problem (Subclause 1.1.12) with the **`asctime`** function. In addition to the caller supplying a pointer to where to store the result, another parameter gives the number of elements in the result array, so that the function does not write past the end of the buffer.

This function is similar to the POSIX **`asctime_r`** function, but that function lacks the parameter giving the size of the result array, and does not perform runtime-constraint checks (like verifying that the calendar year is reasonable).

The format of the string produced by **`asctime`** and **`asctime_s`** is well known, and many programs (and even command scripts) depend upon it. Although the **`strftime`** function provides more flexible formatting, if the exact format of the **`asctime`** result is desired, the **`asctime_s`** will produce it safely with a minimum change to the program (see Subclause 1.1.8).

6.8.2.2 The ctime_s function

The **`ctime_s`** function fixes static internal state problem (Subclause 1.1.12) with the **`ctime`** function. In addition to the caller supplying a pointer to where to store the result, another parameter gives the number of elements in the result array, so that the function does not write past the end of the buffer.

This function is similar to the POSIX **`ctime_r`** function, but that function lacks the parameter giving the size of the result array, and does not perform runtime-constraint checks (like verifying that the calendar year is reasonable).

The format of the string produced by **`ctime`** and **`ctime_s`** is well known, and many programs (and even command scripts) depend upon it. Although the **`strftime`** function provides more flexible formatting, if the exact format of the **`ctime`** result is desired, the **`ctime_s`** will produce it safely with a minimum change to the program (see Subclause 1.1.8).

6.8.2.3 The gmtime_s function

The **`gmtime_s`** function fixes static internal state problem (Subclause 1.1.12) with the **`gmtime`** function.

This function is similar to the POSIX **`gmtime_r`** function, differing only in that **`gmtime_s`** checks runtime-constraints. The committee debated whether the **`gmtime_s`** function should be named **`gmtime_r`**, but decided against it for two reasons. First, the committee wanted all of the function names in the technical report to follow a uniform pattern (Subclause 1.1.13). Second, the runtime-constraint support does make these functions different.

6.8.2.4 The localtime_s function

The **`localtime_s`** function fixes static internal state problem (Subclause 1.1.12) with the **`localtime`** function.

This function is similar to the POSIX **`localtime_r`** function, differing only in that **`localtime_s`** checks runtime-constraints. The committee debated whether the **`localtime_s`** function should be named **`localtime_r`**, but decided against it for two reasons. First, the committee wanted all of the function names in the technical report to follow a uniform pattern (Subclause 1.1.13). Second, the runtime-constraint support does make these functions different.

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

6.9.1 Formatted wide character input/output functions

The rationale for these functions is the same as their multibyte counterparts (Subclause 6.5.3).

6.9.2 General wide string utilities

6.9.2.1 Wide string copying functions

The rationale for these functions is the same as their multibyte counterparts (Subclause 6.7.1).

6.9.2.2 Wide string concatenation functions

The rationale for these functions is the same as their multibyte counterparts (Subclause 6.7.2).

6.9.2.3 Wide string search functions

6.9.2.3.1 The wcstok_s function

The **`wcstok_s`** function has a new parameter, **`s1max`**, that prevents **`wcstok_s`** from storing outside of the wide string being tokenized. (The wide string being divided into tokens is both an input and output of the function since **`wcstok_s`** stores null wide characters into the wide string.)

6.9.2.4 Miscellaneous functions

The rationale for the **`wcsnlen_s`** function is the same as its multibyte counterpart (Subclause 6.7.4.3).

6.9.3 Extended multibyte/wide character conversion utilities

The rationale for these functions is same as the function in Subclauses 6.6.4 and 6.6.5, except that these functions also fix the static internal state problem. The functions in this Subclause should be preferred over the functions in Subclauses 6.6.4 and 6.6.5 when the cost of modifying the program to manage the state is reasonable (see Subclause 1.1.8).