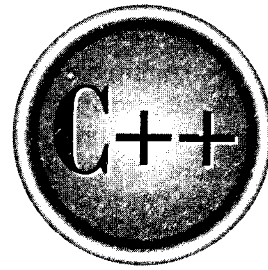
The Complete Reference



Chapter 30

Utility Functions

The standard function library defines several utility functions that provide various commonly used services. They include a number of conversions, variable-length argument processing, sorting and searching, and random number generation. Many of the functions covered here require the use of the header **<cstdlib>**. (A C program must use the header file **stdlib.h**.) In this header are defined **div_t** and **ldiv_t**, which are the types of values returned by **div( )** and **ldiv( )**, respectively. Also defined is the type **size_t**, which is the unsigned value returned by **sizeof**. The following macros are defined:

Macro	Meaning
MB_CUR_MAX	Maximun length (in bytes) of a multibyte character.
NULL	A null pointer.
RAND_MAX	The maximum value that can be returned by the **rand( )** function.
EXIT_FAILURE	The value returned to the calling process if program termination is unsuccessful.
EXIT_SUCCESS	The value returned to the calling process if program termination is successful.

If a function requires a different header than **<cstdlib>**, that function description will discuss it.

abort

```
#include <cstdlib>
void abort(void);
```

The **abort( )** function causes immediate abnormal termination of a program. Generally, no files are flushed. In environments that support it, **abort( )** will return an implementation-defined value to the calling process (usually the operating system) indicating failure.

Related functions are **exit( )** and **atexit( )**.

abs

```
#include <cstdlib>
int abs(int num);
long abs(long num);
double abs(double num);
```

The **abs( )** function returns the absolute value of *num*. The **long** version of **abs( )** is the same as **labs( )**. The **double** version of **abs( )** is the same as **fabs( )**.

A related function is **labs( )**.

assert

```
#include <cassert>
void assert(int exp);
```

The **assert( )** macro, defined in its header **<cassert>**, writes error information to **stderr** and then aborts program execution if the expression *exp* evaluates to zero. Otherwise, **assert( )** does nothing. Although the exact output is implementation defined, many compilers use a message similar to this:

Assertion failed: <*expression*>, file <*file*>, line <*linenum*>

The **assert( )** macro is generally used to help verify that a program is operating correctly, with the expression being devised in such a way that it evaluates to true only when no errors have taken place.

It is not necessary to remove the **assert( )** statements from the source code once a program is debugged because if the macro **NDEBUG** is defined (as anything), the **assert( )** macros will be ignored.

A related function is **abort( )**.

atexit

```
#include <cstdlib>
int atexit(void (*func)(void));
```

The **atexit( )** function causes the function pointed to by *func* to be called upon normal program termination. The **atexit( )** function returns zero if the function is successfully registered as a termination function, nonzero otherwise.

At least 32 termination functions may be established, and they will be called in the reverse order of their establishment.

Related functions are **exit( )** and **abort( )**.

atof

```
#include <cstdlib>
double atof(const char *str);
```

The **atof( )** function converts the string pointed to by *str* into a **double** value. The string must contain a valid floating-point number. If this is not the case, the returned value is undefined.

The number may be terminated by any character that cannot be part of a valid floating-point number. This includes white space, punctuation (other than periods), and characters other than E or e. This means that if **atof( )** is called with "100.00HELLO", the value 100.00 will be returned.

Related functions are **atoi( )** and **atol( )**.

atoi

```
#include <cstdlib>
int atoi(const char *str);
```

The **atoi( )** function converts the string pointed to by *str* into an **int** value. The string must contain a valid integer number. If this is not the case, the returned value is undefined; however, most implementations will return zero.

The number may be terminated by any character that cannot be part of an integer number. This includes white space, punctuation, and characters. This means that if **atoi( )** is called with "123.23", the integer value 123 will be returned, and the ".23" is ignored.

Related functions are **atof( )** and **atol( )**.

atol

```
#include <cstdlib>
long atol(const char *str);
```

The **atol( )** function converts the string pointed to by *str* into a **long** value. The string must contain a valid long integer number. If this is not the case, the returned value is undefined; however, most implementations will return zero.

The number may be terminated by any character that cannot be part of an integer number. This includes white space, punctuation, and characters. This means that if **atol( )** is called with "123.23", the long integer value 123L will be returned, and the ".23" is ignored.

Related functions are **atof( )** and **atoi( )**.

bsearch

```
#include <cstdlib>
void *bsearch(const void *key, const void *buf,
```

```
                size_t num, size_t size,
                int (*compare)(const void *, const void *));
```

The **bsearch( )** function performs a binary search on the sorted array pointed to by *buf* and returns a pointer to the first member that matches the key pointed to by *key*. The number of elements in the array is specified by *num*, and the size (in bytes) of each element is described by *size*.

The function pointed to by *compare* is used to compare an element of the array with the key. The form of the *compare* function must be as follows:

int *func_name*(const void **arg1*, const void **arg2*);

It must return values as described in the following table:

Comparison	Value Returned
arg1 is less than *arg2*	Less than zero
arg1 is equal to *arg2*	Zero
arg1 is greater than *arg2*	Greater than zero

The array must be sorted in ascending order with the lowest address containing the lowest element.

If the array does not contain the key, a null pointer is returned.

A related function is **qsort( )**.

div

```
#include <cstdlib>
div_t div(int numerator, int denominator);
ldiv_t div(long numerator, long denominator);
```

The **int** version of **div( )** returns the quotient and the remainder of the operation *numerator / denominator* in a structure of type **div_t**. The **long** version of **div( )** returns the quotient and remainder in a structure of type **ldiv_t**. The **long** version of **div( )** provides the same capabilities as the **ldiv( )** function.

The structure type **div_t** will have at least these two fields:

```
int quot; /* quotient */
int rem;  /* remainder */
```

The structure type **ldiv_t** will have at least these two fields:

```
long quot; /* quotient */
long rem;  /* remainder */
```

A related function is **ldiv( )**.

exit

```
#include <cstdlib>
void exit(int exit_code);
```

The **exit( )** function causes immediate, normal termination of a program.

The value of *exit_code* is passed to the calling process, usually the operating system, if the environment supports it. By convention, if the value of *exit_code* is zero, or **EXIT_SUCCESS**, normal program termination is assumed. A nonzero value, or **EXIT_FAILURE**, is used to indicate an implementation-defined error.

Related functions are **atexit( )** and **abort( )**.

getenv

```
#include <cstdlib>
char *getenv(const char *name);
```

The **getenv( )** function returns a pointer to environmental information associated with the string pointed to by *name* in the implementation-defined environmental information table. The string returned must never be changed by the program.

The environment of a program may include such things as path names and devices online. The exact nature of this data is implementation defined. You will need to refer to your compiler's documentation for details.

If a call is made to **getenv( )** with an argument that does not match any of the environment data, a null pointer is returned.

A related function is **system( )**.

labs

```
#include <cstdlib>
long labs(long num);
```

The **labs( )** function returns the absolute value of *num*.

A related function is **abs( )**.

ldiv

```
#include <cstdlib>
ldiv_t ldiv(long numerator, long denominator);
```

The **ldiv( )** function returns the quotient and the remainder of the operation *numerator / denominator.*

The structure type **ldiv_t** will have at least these two fields:

```
long quot; /* quotient */
long rem;  /* remainder */
```

A related function is **div( )**.

longjmp

```
#include <csetjmp>
void longjmp(jmp_buf envbuf, int status);
```

The **longjmp( )** function causes program execution to resume at the point of the last call to **setjmp( )**. These two functions provide a means of jumping between functions. Notice that the header **<csetjmp>** is required.

The **longjmp( )** function operates by resetting the stack to the state as described in *envbuf,* which must have been set by a prior call to **setjmp( )**. This causes program execution to resume at the statement following the **setjmp( )** invocation. That is, the computer is "tricked" into thinking that it never left the function that called **setjmp( )**. (As a somewhat graphic explanation, the **longjmp( )** function "warps" across time and (memory) space to a previous point in your program without having to perform the normal function return process.)

The buffer *evnbuf* is of type **jmp_buf**, which is defined in the header **<csetjmp>**. The buffer must have been set through a call to **setjmp( )** prior to calling **longjmp( )**.

The value of *status* becomes the return value of **setjmp( )** and may be interrogated to determine where the long jump came from. The only value that is not allowed is zero.

By far the most common use of **longjmp( )** is to return from a deeply nested set of routines when an error occurs.

A related function is **setjmp( )**.

mblen

```
#include <cstdlib>
int mblen(const char *str, size_t size);
```

The **mblen( )** function returns the length (in bytes) of a multibyte character pointed to by *str*. Only the first *size* number of characters are examined. It returns –1 on error.

If *str* is null, then **mblen( )** returns non-zero if multibyte characters have state-dependent encodings. If they do not, zero is returned.

Related functions are **mbtowc( ) and wctomb( )**.

mbstowcs

```
#include <cstdlib>
size_t mbstowcs(wchar_t *out, const char *in, size_t size);
```

The **mbstowcs( )** function converts the multibyte string pointed to by *in* into a wide character string and puts that result in the array pointed to by *out*. Only *size* number of bytes will be stored in *out*.

The **mbstowcs( )** function returns the number of multibyte characters that are converted. If an error occurs, the function returns –1.

Related functions are **wcstombs( )**, **mbtowc( )**.

mbtowc

```
#include <cstdlib>
int mbtowc(wchar_t *out, const char *in, size_t size);
```

The **mbtowc( )** function converts the multibyte character in the array pointed to by *in* into its wide character equivalent and puts that result in the object pointed to by *out*. Only *size* number of characters will be examined.

This function returns the number of bytes that are put into *out*. –1 is returned if an error occurs. If *in* is null, then **mbtowc( )** returns non-zero if multibyte characters have state-dependent encodings. If they do not, zero is returned.

Related functions are **mblen( )**, **wctomb( )**.

qsort

```
#include <cstdlib>
void qsort(void *buf, size_t num, size_t size,
           int (*compare) (const void *, const void *));
```

The **qsort( )** function sorts the array pointed to by *buf* using a Quicksort (developed by C.A.R. Hoare). The Quicksort is the best general-purpose sorting algorithm. Upon termination, the array will be sorted. The number of elements in the array is specified by *num*, and the size (in bytes) of each element is described by *size*.

The function pointed to by *compare* is used to compare an element of the array with the key. The form of the *compare* function must be as follows:

int *func_name*(const void **arg1*, const void **arg2*);

It must return values as described here:

Comparison	Value Returned
arg1 is less than *arg2*	Less than zero
arg1 is equal to *arg2*	Zero
arg1 is greater than *arg2*	Greater than zero

The array is sorted into ascending order with the lowest address containing the lowest element.

A related function is **bsearch( )**.

raise

```
#include <csignal>
int raise(int signal);
```

The **raise( )** function sends the signal specified by *signal* to the executing program. It returns zero if successful, and nonzero otherwise. It uses the header **<csignal>**.

The following signals are defined by Standard C++. Of course, your compiler is free to provide additional signals.

Macro	Meaning
SIGABRT	Termination error
SIGFPE	Floating-point error
SIGILL	Bad instruction

Macro	Meaning
SIGINT	User pressed CTRL-C
SIGSEGV	Illegal memory access
SIGTERM	Terminate program

A related function is **signal( )**.

rand

```
#include <cstdlib>
int rand(void);
```

The **rand( )** function generates a sequence of pseudorandom numbers. Each time it is called, an integer between zero and **RAND_MAX** is returned.

A related function is **srand( )**.

setjmp

```
#include <csetjmp>
int setjmp(jmp_buf envbuf);
```

The **setjmp( )** function saves the contents of the system stack in the buffer *envbuf* for later use by **longjmp( )**. It uses the header **<csetjmp>**.

The **setjmp( )** function returns zero upon invocation. However, **longjmp( )** passes an argument to **setjmp( )** when it executes, and it is this value (always nonzero) that will appear to be the value of **setjmp( )** after a call to **longjmp( )** has occurred.

See **longjmp** for additional information.

A related function is **longjmp( )**.

signal

```
#include <csignal>
void (*signal(int signal, void (*func)(int))) (int);
```

The **signal( )** function registers the function pointed to by *func* as a handler for the signal specified by *signal*. That is, the function pointed to by *func* will be called when *signal* is received by your program.

The value of *func* may be the address of a signal handler function or one of the following macros, defined in **<csignal>**:

Macro	Meaning
SIG_DFL	Use default signal handling
SIG_IGN	Ignore the signal

If a function address is used, the specified handler will be executed when its signal is received.

On success, **signal( )** returns the address of the previously defined function for the specified signal. On error, **SIG_ERR** (defined in **<csignal>**) is returned.

A related function is **raise( )**.

srand

```
#include <cstdlib>
void srand(unsigned seed);
```

The **srand( )** function is used to set a starting point for the sequence generated by **rand( )**. (The **rand( )** function returns pseudorandom numbers.)

srand( ) is generally used to allow multiple program runs to use different sequences of pseudorandom numbers by specifying different starting points. Conversely, you can also use **srand( )** to generate the same pseudorandom sequence over and over again by calling it with the same seed before obtaining each sequence.

A related function is **rand( )**.

strtod

```
#include <cstdlib>
double strtod(const char *start, char **end);
```

The **strtod( )** function converts the string representation of a number stored in the string pointed to by *start* into a **double** and returns the result.

The **strtod( )** function works as follows. First, any white space in the string pointed to by *start* is stripped. Next, each character that comprises the number is read. Any character that cannot be part of a floating-point number will cause this process to stop. This includes white space, punctuation (other than periods), and characters other than E or e. Finally, *end* is set to point to the remainder, if any, of the original string. This means that if **strtod( )** is called with "100.00 Pliers", the value 100.00 will be returned, and *end* will point to the space that precedes "Pliers".

If no conversion takes place, zero is returned. If overflow occurs, **strtod( )** returns either **HUGE_VAL** or **-HUGE_VAL** (indicating positive or negative overflow), and the global variable **errno** is set to **ERANGE**, indicating a range error. If underflow occurs, then zero is returned and the global variable **errno** is set to **ERANGE**.

A related function is **atof( )**.

strtol

```
#include <cstdlib>
long strtol(const char *start, char **end,
            int radix);
```

The **strtol( )** function converts the string representation of a number stored in the string pointed to by *start* into a **long** and returns the result. The base of the number is determined by *radix*. If *radix* is zero, the base is determined by rules that govern constant specification. If *radix* is other than zero, it must be in the range 2 through 36.

The **strtol( )** function works as follows. First, any white space in the string pointed to by *start* is stripped. Next, each character that comprises the number is read. Any character that cannot be part of a long integer number will cause this process to stop. This includes white space, punctuation, and characters. Finally, *end* is set to point to the remainder, if any, of the original string. This means that if **strtol( )** is called with "100 Pliers", the value 100L will be returned, and *end* will point to the space that precedes "Pliers".

If the result cannot be represented by a long integer, **strtol( )** returns either **LONG_MAX** or **LONG_MIN** and the global **errno** is set to **ERANGE**, indicating a range error. If no conversion takes place, zero is returned.

A related function is **atol( )**.

strtoul

```
#include <cstdlib>
unsigned long strtoul(const char *start, char **end,
                      int radix);
```

The **strtoul( )** function converts the string representation of a number stored in the string pointed to by *start* into an **unsigned long** and returns the result. The base of the number is determined by *radix*. If *radix* is zero, the base is determined by rules that govern constant specification. If the radix is specified, it must be in the range 2 through 36.

The **strtoul( )** function works as follows. First, any white space in the string pointed to by *start* is stripped. Next, each character that comprises the number is read. Any character that cannot be part of an unsigned long integer number will cause this process to stop. This includes white space, punctuation, and characters. Finally, *end* is set to point to the remainder, if any, of the original string. This means that if **strtoul( )** is called with " 100 Pliers", the value 100L will be returned, and *end* will point to the space that precedes "Pliers".

If the result cannot be represented by an unsigned long integer, **strtoul( )** returns **ULONG_MAX** and the global variable **errno** is set to **ERANGE**, indicating a range error. If no conversion takes place, zero is returned.

A related function is **strtol( )**.

system

```
#include <cstdlib>
int system(const char *str);
```

The **system( )** function passes the string pointed to by *str* as a command to the command processor of the operating system.

If **system( )** is called with a null pointer, it will return nonzero if a command processor is present, and zero otherwise. (Some C++ code will be executed in dedicated systems that do not have operating systems and command processors, so you may not be able to assume that a command processor is present.) The return value of **system( )** is implementation defined. However, generally it will return zero if the command was successfully executed, and nonzero otherwise.

A related function is **exit( )**.

va_arg, va_start, and va_end

```
#include <cstdarg>
type va_arg(va_list argptr, type);
void va_end(va_list argptr);
void va_start(va_list argptr, last_parm);
```

The **va_arg( )**, **va_start( )**, and **va_end( )** macros work together to allow a variable number of arguments to be passed to a function. The most common example of

a function that takes a variable number of arguments is **printf( )**. The type **va_list** is defined by <**cstdarg**>.

The general procedure for creating a function that can take a variable number of arguments is as follows. The function must have at least one known parameter, but may have more, prior to the variable parameter list. The rightmost known parameter is called the *last_parm*. The name of *last_parm* is used as the second parameter in a call to **va_start( )**. Before any of the variable-length parameters can be accessed, the argument pointer *argptr* must be initialized through a call to **va_start( )**. After that, parameters are returned via calls to **va_arg( )**, with *type* being the type of the next parameter. Finally, once all the parameters have been read and prior to returning from the function, a call to **va_end( )** must be made to ensure that the stack is properly restored. If **va_end( )** is not called, a program crash is very likely.

A related function is **vprintf( )**.

wcstombs

```
#include <cstdlib>
size_t wcstombs(char *out, const wchar_t *in, size_t size);
```

The **wcstombs( )** converts the wide-character array pointed to by *in* into its multibyte equivalent and puts the result in the array pointed to by *out*. Only the first *size* bytes of *in* are converted. Conversion stops before that if the null terminator is encountered.

If successful, **wcstombs( )** returns the number of bytes converted. On failure, –1 is returned.

Related functions are **wctomb( )** and **mbstowcs( )**.

wctomb

```
#include <cstdlib>
int wctomb(char *out,  wchar_t in);
```

The **wctomb( )** converts the wide character in *in* into its multibyte equivalent and puts the result in the object pointed to by *out*. The array pointed to by *out* must be at least **MB_CUR_MAX** characters long.

If successful, **wctomb( )** returns the number of bytes contained in the multibyte character. On failure, –1 is returned.

If *out* is null, then **wctomb( )** returns nonzero if the multibyte character has state-dependent encodings and zero if it does not.

Related functions are **wcstombs( )** and **mbtowc( )**.