
5.8. Header Files `<cstddef>`, `<cstdlib>`, and `<cstring>`

The following header files compatible with C are often used in C++ programs: `<cstddef>`, `<cstdlib>`, and `<cstring>`. They are the C++ versions of the C header files `<stddef.h>`, `<stdlib.h>`, and `<string.h>` and they define some common constants, macros, types, and functions.

5.8.1. Definitions in `<cstddef>`

Table 5.26 shows the definitions of the `<cstddef>` header file. Before C++11, `NULL` was often used to indicate that a pointer points to nothing. Since C++11, `nullptr` is provided for this semantics (see Section 3.1.1, page 14).

Table 5.26. Definitions in `<cstddef>`

Identifier	Meaning
`NULL`	Pointer value for "not defined" or "no value"
`nullptr_t`	Type of `nullptr` (since C++11)
`size_t`	Unsigned type for size units, such as number of elements
`ptrdiff_t`	Signed type for differences of pointer
`max_align_t`	Type with maximum alignment in all contexts (since C++11)
`offsetof(`*type*`,`*mem*`)`	Offset of a member *mem* in a structure or union *type*

Note that `NULL` in C++ is guaranteed to be simply the value `0` (either as an `int` or as a `long`). In C, `NULL` is often defined as `(void*)0`. However, this is incorrect in C++, which requires that the type of `NULL` be an integer type. Otherwise, you could not assign `NULL` to a pointer. This is because in C++ there is no automatic conversion from `void*` to any other type. Since C++11, you should use `nullptr` instead (see Section 3.1.1, page 14).[31] Note also that `NULL` is also defined in the header files `<cstdio>`, `<cstdlib>`, `<cstring>`, `<ctime>`, `<cwchar>`, and `<clocale>`.

[31] Due to the mess with the type of `NULL`, several people and style guides recommend not using `NULL` in C++. Instead, `0` or a special user-defined constant, such as `NIL`, might work better. Fortunately, this problem is solved with `nullptr`.

5.8.2. Definitions in `<cstdlib>`

Table 5.27 shows the most important definitions of the `<cstdlib>` header file. The two constants `EXIT_SUCCESS` and `EXIT_FAILURE` are defined as arguments for `exit()` and can also be used as a return value in `main()`.

Table 5.27. Definitions in `<cstdlib>`

Definition	Meaning
`EXIT_SUCCESS`	Indicates a normal end of the program
`EXIT_FAILURE`	Indicates an abnormal end of the program
`exit (int` *status*`)`	Exit program (cleans up static objects)
`quick_exit (int` *status*`)`	Exit program with cleanup according to `at_quick_exit()` (since C++11)
`_Exit (int` *status*`)`	Exit program with no cleanup (since C++11)
`abort()`	Abort program (might force a crash on some systems)
`atexit (void (*`*func*`)())`	Call *func* on exit
`at_quick_exit (void (*`*func*`)())`	Call *func* on `quick_exit()` (since C++11)

The functions that are registered by `atexit()` are called at normal program termination in reverse order of their registration. It doesn't matter whether the program exits due to a call of `exit()` or the end of `main()`. No arguments are passed.

The `exit()` and `abort()` functions are provided to terminate a program in any function without going back to `main()`:

- `exit()` destroys all static objects, flushes all buffers, closes all I/O channels, and terminates the program, including calling `atexit()` functions. If functions passed to `atexit()` throw exceptions, `terminate()` is called.
- `abort()` terminates a program immediately with no cleanup.

Neither of these functions destroys local objects, because no stack unwinding occurs. To ensure that the destructors of all local objects are called, you should use exceptions or the ordinary return mechanism to return to and exit `main()`.

Since C++11, the `quick_exit()` semantics provided does not destroy objects but calls functions registered by calls to `at_quick_exit()` in the reverse order of their registration and calls `_Exit()`, which terminates the program then without any destruction or cleanup.[32] This means that `quick_exit()` and `_Exit()` do not flush standard file buffers (standard output and error output).

[32] This feature was introduced to avoid the risk that detached threads access global/static objects (see Section 18.2.1, page 967).

The usual way for C++ to abort programs — which is an unexpected end in contrast to an expected end signaling an error — is to call `std::terminate()`, which by default calls `abort()`. This is done, for example, if a destructor or a function declared with `noexcept` (see Section 3.1.7, page 24) throws.

5.8.3. Definitions in `<cstring>`

Table 5.28 shows the most important definitions of the `<cstring>` header file: the low-level functions to set, copy, and move memory. One application of these functions is character traits (see Section 16.1.4, page 855).

Table 5.28. Definitions in `<cstring>`

Definition	Meaning
`memchr (const void* ptr, int c, size_t len)`	Finds character *c* in first *len* bytes of *ptr*
`memcmp (const void* ptr1, const void* ptr2, size_t len)`	Compares *len* bytes of *ptr1* and *ptr2*
`memcpy (void* toPtr, const void* fromPtr, size_t len)`	Copies *len* bytes of *fromPtr* to *toPtr*
`memmove (void* toPtr, const void* fromPtr, size_t len)`	Copies *len* bytes of *fromPtr* to *toPtr* (areas may overlap)
`memset (void* ptr, int c, size_t len)`	Assigns character *c* to first *len* bytes of *ptr*