std::signal

Defined in header <csignal>

```
/*signal-handler*/* signal(int sig, /*signal-handler*/* handler);      (1)
extern "C" using /*signal-handler*/ = void(int); // exposition-only    (2)
```

Sets the handler for signal sig. The signal handler can be set so that default handling will occur, signal is ignored, or a user-defined function is called.

When signal handler is set to a function and a signal occurs, it is implementation defined whether `std::signal(sig, SIG_DFL)` will be executed immediately before the start of signal handler. Also, the implementation can prevent some implementation-defined set of signals from occurring while the signal handler runs.

For some of the signals, the implementation may call `std::signal(sig, SIG_IGN)` at the startup of the program. For the rest, the implementation must call `std::signal(sig, SIG_DFL)`.

(Note: POSIX introduced sigaction (http://pubs.opengroup.org/onlinepubs/9699919799/functions/sigaction.html) to standardize these implementation-defined behaviors)

Parameters

sig - the signal to set the signal handler to. It can be an implementation-defined value or one of the following values:

SIGABRT	
SIGFPE	
SIGILL	defines signal types
SIGINT	(macro constant)
SIGSEGV	
SIGTERM	

handler - the signal handler. This must be one of the following:

- SIG_DFL macro. The signal handler is set to default signal handler.
- SIG_IGN macro. The signal is ignored.
- pointer to a function. The signature of the function must be equivalent to the following:

```
extern "C" void fun(int sig);
```

Return value

Previous signal handler on success or SIG_ERR on failure (setting a signal handler can be disabled on some implementations).

Signal handler

The following limitations are imposed on the user-defined function that is installed as a signal handler.

If the signal handler is called NOT as a result of std::abort or std::raise (asynchronous signal), the behavior is undefined if

- the signal handler calls any function within the standard library, except
  - std::abort
  - std::_Exit
  - std::quick_exit

  (until C++17)
  - `std::signal` with the first argument being the number of the signal currently handled (async handler can re-register itself, but not other signals).
- the signal handler refers to any object with static storage duration that is not std::atomic(since C++11) or `volatile std::sig_atomic_t`.

The behavior is undefined if any signal handler performs any of the following: (since C++17)

- call to any library function, except the following *signal-safe* functions (note, in particular, dynamic allocation is not signal-safe):

- members functions of `std::atomic` and non-member functions from `<atomic>` if the atomic type they operate on is lock-free. The functions `std::atomic_is_lock_free` and `std::atomic::is_lock_free` are signal-safe for any atomic type.
- `std::signal` with the first argument being the number of the signal currently handled (signal handler can re-register itself, but not other signals).
- member functions of `std::numeric_limits`
- `std::_Exit`
- `std::abort`
- `std::quick_exit`
- The member functions of `std::initializer_list` and the `std::initializer_list` overloads of `std::begin` and `std::end`
- `std::forward`, `std::move`, `std::move_if_noexcept`
- All functions from `<type_traits>`
- `std::memcpy` and `std::memmove`

- access to an object with thread storage duration
- a `dynamic_cast` expression
- a `throw` expression
- entry to a `try` block, including function-try-block
- initialization of a static variable that performs dynamic non-local initialization (including delayed until first ODR-use)
- waits for completion of initialization of any variable with static storage duration due to another thread concurrently initializing it

If the user defined function returns when handling `SIGFPE`, `SIGILL`, `SIGSEGV` or any other implementation-defined signal specifying a computational exception, the behavior is undefined.

If the signal handler is called as a result of `std::abort` or `std::raise` (synchronous signal), the behavior is undefined if the signal handler calls `std::raise`.

On entry to the signal handler, the state of the floating-point environment and the values of all objects is unspecified, except for

- objects of type `volatile std::sig_atomic_t`
- objects of lock-free `std::atomic` types (since C++11) (until C++14)
- side effects made visible through `std::atomic_signal_fence` (since C++11)

On return from a signal handler, the value of any object modified by the signal handler that is not `volatile std::sig_atomic_t` or lock-free `std::atomic` is indeterminate.

A call to the function `signal()` synchronizes-with any resulting invocation of the signal handler.

If a signal handler is executed as a result of a call to `std::raise` (synchronously), then the execution of the handler is *sequenced-after* the invocation of `std::raise` and *sequenced-before* the return from it and runs on the same thread as `std::raise`. Execution of the handlers for other signals is *unsequenced* with respect to the rest of the program and runs on an unspecified thread.

Two accesses to the same object of type `volatile std::sig_atomic_t` do not result in a data race if both occur in the same thread, even if one or more occurs in a signal handler. For each signal handler invocation, evaluations performed by the thread invoking a signal handler can be divided into two groups A and B, such that no evaluations in B *happen-before* evaluations in A, and the evaluations of such `volatile std::sig_atomic_t` objects take values as though all evaluations in A happened-before the execution of the signal handler and the execution of the signal handler *happened-before* all evaluations in B. (since C++14)

Notes

POSIX requires that `signal` is thread-safe, and specifies a list of async-signal-safe library functions (http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_04) that may be called from any signal handler.

Signal handlers are expected to have C linkage and, in general, only use the features from the common subset of C and C++. It is implementation-defined if a function with C++ linkage can be used as a signal handler. (until C++17)

There is no restriction on the linkage of signal handlers. (since C++17)

Example

Run this code

```
#include <csignal>
#include <iostream>

namespace
{
  volatile std::sig_atomic_t gSignalStatus;
}

void signal_handler(int signal)
{
  gSignalStatus = signal;
}

int main()
{
  // Install a signal handler
  std::signal(SIGINT, signal_handler);

  std::cout << "SignalValue: " << gSignalStatus << '\n';
  std::cout << "Sending signal " << SIGINT << '\n';
  std::raise(SIGINT);
  std::cout << "SignalValue: " << gSignalStatus << '\n';
}
```

Possible output:

```
SignalValue: 0
Sending signal 2
SignalValue: 2
```

See also

raise	runs the signal handler for particular signal (function)
atomic_signal_fence (C++11)	fence between a thread and a signal handler executed in the same thread (function)
C documentation for **signal**	

Retrieved from "https://en.cppreference.com/mwiki/index.php?title=cpp/utility/program/signal&oldid=138667"