Chapter 31: Signal handling

Parameter	Details
sig	The signal to set the signal handler to, one of `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGTERM`, `SIGINT`, `SIGSEGV` or some implementation defined value
func	The signal handler, which is either of the following: `SIG_DFL`, for the default handler, `SIG_IGN` to ignore the signal, or a function pointer with the signature `void foo(int sig);`.

Section 31.1: Signal Handling with “signal()”

Signal numbers can be synchronous (like `SIGSEGV` – segmentation fault) when they are triggered by a malfunctioning of the program itself or asynchronous (like `SIGINT` - interactive attention) when they are initiated from outside the program, e.g by a keypress as `Cntrl-C`.

The `signal()` function is part of the ISO C standard and can be used to assign a function to handle a specific signal

```
#include <stdio.h>  /* printf() */
#include <stdlib.h> /* abort()  */
#include <signal.h> /* signal() */

void handler_nonportable(int sig)
{
    /* undefined behavior, maybe fine on specific platform */
    printf("Catched: %d\n", sig);

    /* abort is safe to call */
    abort();
}

sig_atomic_t volatile finished = 0;

void handler(int sig)
{
    switch (sig) {
    /* hardware interrupts should not return */
    case SIGSEGV:
    case SIGFPE:
    case SIGILL:
```

Version ≥ C11

```
      /* quick_exit is safe to call */
      quick_exit(EXIT_FAILURE);
```

Version < C11

```
      /* use _Exit in pre-C11 */
      _Exit(EXIT_FAILURE);
```

```
    default:
       /* Reset the signal to the default handler,
          so we will not be called again if things go
          wrong on return. */
      signal(sig, SIG_DFL);
      /* let everybody know that we are finished */
      finished = sig;
      return;
   }
}

int main(void)
{
```

```
    /* Catch the SIGSEGV signal, raised on segmentation faults (i.e NULL ptr access */
    if (signal(SIGSEGV, &handler) == SIG_ERR) {
        perror("could not establish handler for SIGSEGV");
        return EXIT_FAILURE;
    }

    /* Catch the SIGTERM signal, termination request */
    if (signal(SIGTERM, &handler) == SIG_ERR) {
        perror("could not establish handler for SIGTERM");
        return EXIT_FAILURE;
    }

    /* Ignore the SIGINT signal, by setting the handler to `SIG_IGN`. */
    signal(SIGINT, SIG_IGN);


    /* Do something that takes some time here, and leaves
       the time to terminate the program from the keyboard. */

    /* Then: */

    if (finished) {
       fprintf(stderr, "we have been terminated by signal %d\n", (int)finished);
        return EXIT_FAILURE;
    }


    /* Try to force a segmentation fault, and raise a SIGSEGV */
    {
      char* ptr = 0;
      *ptr = 0;
    }

    /* This should never be executed */
    return EXIT_SUCCESS;
}
```

Using `signal()` imposes important limitations what you are allowed to do inside the signal handlers, see the remarks for further information.

POSIX recommends the usage of `sigaction()` instead of `signal()`, due to its underspecified behavior and significant implementation variations. POSIX also defines many more signals than ISO C standard, including `SIGUSR1` and `SIGUSR2`, which can be used freely by the programmer for any purpose.