Table 24.1
Signals

Name	*Meaning*
`SIGABRT`	Abnormal termination (possibly caused by a call of `abort`)
`SIGFPE`	Error during an arithmetic operation (possibly division by zero or overflow)
`SIGILL`	Invalid instruction
`SIGINT`	Interrupt
`SIGSEGV`	Invalid storage access
`SIGTERM`	Termination request

The `signal` Function

```
void (*signal(int sig, void (*func)(int)))(int);
```

signal

`<signal.h>` provides two functions: `raise` and `signal`. We'll start with `signal`, which installs a signal-handling function for use later if a given signal should occur. `signal` is much easier to use than you might expect from its rather intimidating prototype. Its first argument is the code for a particular signal; the second argument is a pointer to a function that will handle the signal if it's raised later in the program. For example, the following call of `signal` installs a handler for the `SIGINT` signal:

```
signal(SIGINT, handler);
```

`handler` is the name of a signal-handling function. If the `SIGINT` signal occurs later during program execution, `handler` will be called automatically.

Every signal-handling function must have an `int` parameter and a return type of `void`. When a particular signal is raised and its handler is called, the handler will be passed the code for the signal. Knowing which signal caused it to be called can be useful for a signal handler; in particular, it allows us to use the same handler for several different signals.

A signal-handling function can do a variety of things. Possibilities include ignoring the signal, performing some sort of error recovery, or terminating the program. Unless it's invoked by `abort` or `raise`, however, a signal handler shouldn't call a library function or attempt to use a variable with static storage duration. (There are a few exceptions to these rules, however.)

abort function ➤ 26.2
static storage duration ➤ 18.2
Q&A

If a signal-handling function returns, the program resumes executing from the point at which the signal occurred, except in two cases: (1) If the signal was `SIGABRT`, the program will terminate (abnormally) when the handler returns. (2) The effect of returning from a function that has handled `SIGFPE` is undefined. (In other words, don't do it.)

Although `signal` has a return value, it's often discarded. The return value, a pointer to the previous handler for the specified signal, can be saved in a variable if desired. In particular, if we plan to restore the original signal handler later, we need to save `signal`'s return value:

```
void (*orig_handler)(int);    /* function pointer variable */
...
```