that should happen, the function will return a ***null pointer***. A null pointer is a "pointer to nothing"—a special value that can be distinguished from all valid pointers. After we've stored the function's return value in a pointer variable, we must test to see if it's a null pointer.

It's the programmer's responsibility to test the return value of any memory allocation function and take appropriate action if it's a null pointer. The effect of attempting to access memory through a null pointer is undefined: the program may crash or behave unpredictably.

**Q&A** The null pointer is represented by a macro named NULL, so we can test `malloc`'s return value in the following way:

```
p = malloc(10000);
if (p == NULL) {
  /* allocation failed; take appropriate action */
}
```

Some programmers combine the call of `malloc` with the NULL test:

```
if ((p = malloc(10000)) == NULL) {
  /* allocation failed; take appropriate action */
}
```

**C99** The NULL macro is defined in six headers: `<locale.h>`, `<stddef.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, and `<time.h>`. (The C99 header `<wchar.h>` also defines NULL.) As long as one of these headers is included in a program, the compiler will recognize NULL. A program that uses any of the memory allocation functions will include `<stdlib.h>`, of course, making NULL available.

In C, pointers test true or false in the same way as numbers. All non-null pointers test true; only null pointers are false. Thus, instead of writing

```
if (p == NULL) ...
```

we could write

```
if (!p) ...
```

and instead of writing

```
if (p != NULL) ...
```

we could write

```
if (p) ...
```

As a matter of style, I prefer the explicit comparison with NULL.