A: Not necessarily. Each C compiler is allowed to represent null pointers in a different way, and not all compilers use a zero address. For example, some compilers use a nonexistent memory address for the null pointer; that way, attempting to access memory through a null pointer can be detected by the hardware.

How the null pointer is stored inside the computer shouldn't concern us; that's a detail for compiler experts to worry about. The important thing is that, when used in a pointer context, 0 is converted to the proper internal form by the compiler.

Q: Is it acceptable to use `NULL` as a null character?

A: Definitely not. `NULL` is a macro that represents the null *pointer*, not the null *character*. Using `NULL` as a null character will work with some compilers, but not with all (since some define `NULL` as `(void *) 0`). In any event, using `NULL` as anything other than a pointer can lead to a great deal of confusion. If you want a name for the null character, define the following macro:

```
#define NUL '\0'
```

***Q: When my program terminates, I get the message *"Null pointer assignment."* What does this mean?**

A: This message, which is produced by programs compiled with some older DOS-based C compilers, indicates that the program has stored data in memory using a bad pointer (but not necessarily a null pointer). Unfortunately, the message isn't displayed until the program terminates, so there's no clue as to which statement caused the error. The *"Null pointer assignment"* message can be caused by a missing `&` in `scanf`:

```
scanf("%d", i);    /* should have been scanf("%d", &i); */
```

Another possibility is an assignment involving a pointer that's uninitialized or null:

```
*p = i;    /* p is uninitialized or null */
```

***Q: How does a program know that a "null pointer assignment" has occurred?**

A: The message depends on the fact that, in the small and medium memory models, data is stored in a single segment, with addresses beginning at 0. The compiler leaves a "hole" at the beginning of the data segment—a small block of memory that's initialized to 0 but otherwise isn't used by the program. When the program terminates, it checks to see if any data in the "hole" area is nonzero. If so, it must have been altered through a bad pointer.

Q: Is there any advantage to casting the return value of `malloc` or the other memory allocation functions? [p. 416]

A: Not usually. Casting the `void *` pointer that these functions return is unnecessary, since pointers of type `void *` are automatically converted to any pointer type upon assignment. The habit of casting the return value is a holdover from older versions of C, in which the memory allocation functions returned a `char *` value, making the cast necessary. Programs that are designed to be compiled as C++ code