***Q:** **Why does C bother to provide type definitions? Isn't defining a `BOOL` macro just as good as defining a `Bool` type using `typedef`? [p. 149]**

A: There are two important differences between type definitions and macro definitions. First, type definitions are more powerful than macro definitions. In particular, array and pointer types can't be defined as macros. Suppose that we try to use a macro to define a "pointer to integer" type:

```
#define PTR_TO_INT int *
```

The declaration

```
PTR_TO_INT p, q, r;
```

will become

```
int * p, q, r;
```

after preprocessing. Unfortunately, only `p` is a pointer; `q` and `r` are ordinary integer variables. Type definitions don't have this problem.

Second, `typedef` names are subject to the same scope rules as variables; a `typedef` name defined inside a function body wouldn't be recognized outside the function. Macro names, on the other hand, are replaced by the preprocessor wherever they appear.

***Q:** **You said that compilers "can usually determine the value of a `sizeof` expression." Can't a compiler *always* determine the value of a `sizeof` expression? [p. 151]**

A: In C89, yes. In C99, however, there's one exception. The compiler can't determine the size of a variable-length array, because the number of elements in the array may change during the execution of the program.

variable-length arrays ➤8.3

Exercises

Section 7.1

1. Give the decimal value of each of the following integer constants.
   (a) `077`
   (b) `0x77`
   (c) `0XABC`

Section 7.2

2. Which of the following are not legal constants in C? Classify each legal constant as either integer or floating-point.
   (a) `010E2`
   (b) `32.1E+5`
   (c) `0790`
   (d) `100_000`
   (e) `3.978e-2`