or

```
if (!flag) …
```

Carrying this idea one step further, we might even define a macro that can be used as a type:

```
#define BOOL int
```

`BOOL` can take the place of `int` when declaring Boolean variables:

```
BOOL flag;
```

It's now clear that `flag` isn't an ordinary integer variable, but instead represents a Boolean condition. (The compiler still treats `flag` as an `int` variable, of course.) In later chapters, we'll discover better ways to set up a Boolean type in C89 by using type definitions and enumerations.

type definitions ➤7.5
enumerations ➤16.5

C99

Boolean Values in C99

Q&A

The longstanding lack of a Boolean type has been remedied in C99, which provides the `_Bool` type. In this version of C, a Boolean variable can be declared by writing

```
_Bool flag;
```

unsigned integer types ➤7.1

`_Bool` is an integer type (more precisely, an *unsigned* integer type), so a `_Bool` variable is really just an integer variable in disguise. Unlike an ordinary integer variable, however, a `_Bool` variable can only be assigned 0 or 1. In general, attempting to store a nonzero value into a `_Bool` variable will cause the variable to be assigned 1:

```
flag = 5;   /* flag is assigned 1 */
```

It's legal (although not advisable) to perform arithmetic on `_Bool` variables; it's also legal to print a `_Bool` variable (either 0 or 1 will be displayed). And, of course, a `_Bool` variable can be tested in an `if` statement:

```
if (flag)   /* tests whether flag is 1 */
  …
```

`<stdbool.h>` header ➤21.5

In addition to defining the `_Bool` type, C99 also provides a new header, `<stdbool.h>`, that makes it easier to work with Boolean values. This header provides a macro, `bool`, that stands for `_Bool`. If `<stdbool.h>` is included, we can write

```
bool flag;   /* same as _Bool flag; */
```

The `<stdbool.h>` header also supplies macros named `true` and `false`, which stand for 1 and 0, respectively, making it possible to write