```
    if (n < 0)
        num_neg += 1;
```

```
    if (scanf("%d%d%d", &n, &m, &r) != 3) {
        printf("scanf failed to read three items\n");
        exit(EXIT_FAILURE);
    }
```

```
    if (year%4==0 && (year%100!=0 || year%400==0)) {
        /* need to allow for leap years */
        length_of_year = 366;
        length_of_feb = 29;
    } else {
        /* not a leap year */
        length_of_year = 365;
        length_of_feb = 28;
    }
```

```
    if (month==2) {
        length_of_month = length_of_feb;
    } else if (month==4 || month==6 ||
               month==9 || month==11) {
        /* thirty days hath september, april, june,
            and november */
        length_of_month = 30;
    } else {
        /* all the rest have 31, except february... */
        length_of_month = 31;
    }
```

Figure 3.2: Examples of the use of the `if` statement. Statements between a "{" and "}" pair are a compound statement, and treated as a single statement.

At face value, when the number zero is entered the program should print a message that more students can be accepted. But when the program is executed (the lower box in Figure 3.3), the other message is printed, and it is clear that the `if` statement took the first of the two alternative paths. Why? Well, you were warned about this earlier – look carefully at the guard in the `if`, and count the number of "=" characters. Two are required for an equality test, and when there is only one, it is an assignment statement. So that guard says "assign the value `MAX_CLASS_SIZE` to the variable `class_size`, and then, if the value that was assigned is non-zero, execute the first branch of the `if` statement".

Some – unfortunately, not all – C compilers allow you to check for this kind of problem. The `gcc` compiler used while preparing this book does not naturally give such warnings (see the first compilation line in the lower box of Figure 3.3), but can be instructed to look for potential problems via the use of the "`all`" option to the "`-W`" compiler warnings-level flag (the second compilation line), which asks for all warning messages to be listed. The `-ansi` flag similarly asks that the `gcc` compiler

```
#define MAX_CLASS_SIZE 50

    int class_size;
    printf("Enter class size: ");
    scanf("%d", &class_size);
    if (class_size = MAX_CLASS_SIZE) {
        printf("Class is now full\n");
    } else {
        printf("More students can be accepted\n");
    }
```

```
mac: gcc -o equalinif equalinif.c
mac: ./equalinif
Enter class size: 0
Class is now full
mac: gcc -Wall -ansi -o equalinif equalinif.c
equalinif.c: In function 'main':
equalinif.c:11: warning: suggest parentheses around
    assignment used as truth value
```

Figure 3.3: A flawed `if` statement. The program always prints "Class is now full", even if a `class_size` of 0 is entered. The reason why is revealed in the second compilation line shown in the lower box, when all warning messages are requested using "`-Wall`". Now the compiler draws attention to the assignment statement inside the guard of the `if` statement, and suggests that it might be problematic, despite that fact that the program is technically correct as it stands.

```
    int x=3, y=4, z=6;
    if (x>2)
        if (y>6)
            z = 7;
    else
        z = 8;
    printf("After the if statement z=%d\n", z);
```

Figure 3.4: Another flawed `if` statement.

not accept any constructs that are not part of the ANSI standard definition of C. There is no reason at all why you shouldn't *always* use these warning and diagnostic facilities if they are available.

What about the program fragment in Figure 3.4? What is the final value of `z`? Should be 6, right? The first guard is true, but the second guard is false, right? So `z` remains unchanged, right? Wrong! The rule is that an `else` part attaches to the most recent unmatched `if` that is available to it, irrespective of program layout. So the `else` in this fragment belongs to the second `if`, and when the second guard is false, the statement `z=8` is executed.

Like the previous pitfall, the program in Figure 3.4 is correct according to the rules of C, and when compiled using the default `gcc` options, no warning message is