Decision Making in Programs

A computer can carry out only those instructions that are clear and precise.

The computer will always do exactly what you say. That's both good and bad news. The problem is that the computer follows instructions even if they are absurd; it has no judgment with which to question anything. Once again, this is one of the cardinal rules of programming...maybe *the* cardinal rule:

A computer can carry out only those instructions that are absolutely clear.

A computer has no such thing as discretion or judgment. It can follow only those rules that are mathematically precise, such as comparing two values to see whether they are equal.

Interlude: What about Artificial Intelligence (AI)?

"But," I hear you say, "computers are smart! They *can* use judgment. What about that program Big Blue that beat Gary Kasparov, the world's chess champion, in 1997?"

Artificial intelligence (AI) is the exception that proves the rule. It might seem that an AI computer program is exercising judgment...but only because of a complex program made up of thousands, even millions, of individual little decisions, each of which is simple and mathematically clear.

Consider the human brain. The operation of each neuron is simple: Given sufficient stimulus, it fires; otherwise, it doesn't. It's the *network* of billions of neurons, analogous to the millions of lines in a big computer program, that creates a host for consciousness.

Is that so different? This question leads us to Very Big Dilemmas. If a computer can be conscious, is it murder to shut it off or junk it? But if a computer cannot be conscious, then what is it about the brain that's so special? Answering such questions is far outside the scope of this book, but I consider this issue—could robots or computers theoretically become conscious?—the most central problem in philosophy today.

if and if-else

The simplest way to program behavior is to say, "If A is true, then do B." That's what the C++ **if** statement does. Here is the simple form of the **if** statement syntax:


```
if (condition)
    statement
```

There are more complex forms of this statement, which we'll get to soon. But first consider an **if** statement that compares two variables, x and y.

```
if (x == y)
     cout << "x and y are equal.";
```

This is strange. There are two equal signs (==) here instead of one (=). This is not a typo. C++ has two separate operators in this regard: One equal sign means assignment, which copies values into a variable. Two equal signs (==) test for equality.

Note

Using assignment (=) where you meant to use test-for-equality (==) is one of the most common C++ programming errors.

What if, instead of executing one statement in response to a condition, you want to do a series of things? The answer is you can use a *compound statement* (or *block*):

```
if (x == y) {
     cout << "x and y are equal." << endl;
     cout << "Isn't that nice?";
     they_are_equal = true;
}
```

The significance of the block is that either all these statements are executed or none of them is executed. If the condition is not true, the program jumps past the end of the block. A block can be plugged into the **if** statement syntax because of another cardinal rule:

Anywhere you can use a statement in C++ syntax, you can use a compound statement (or *block*).

A compound statement is just another kind of statement. The block itself is not terminated by a semicolon (;)—only the statements inside are. This is one of those few exceptions to the semicolon rule.

Here's the **if** statement syntax again:

```
if (condition)
    statement
```

Applying the cardinal rule I just stated, we can insert a block for the *statement*:

```
if (condition) {
    statements
}
```

where *statements* is zero or more statements (there will almost always be at least one).

You can also specify actions to take if the condition is *not* true. This is optional. As you might guess, this variation uses the **else** keyword:


```
if (condition)
    statement1
else
    statement2
```

Either *statement1* or *statement2*, or both, can be a compound statement. Now you have the complete **if** syntax.

Here's a brief example:

```
if (x == y)
    cout << "x and y are equal";
else
    cout << "x and y are NOT equal";
```

This code can be rewritten to use statement-block style:

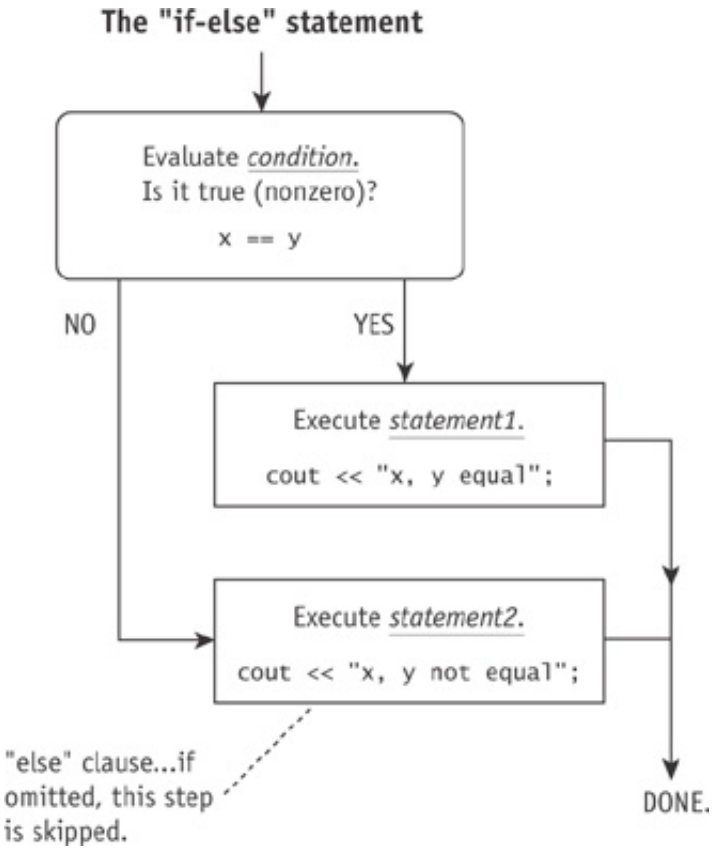
```
if (x == y) {
    cout << "x and y are equal";
} else {
    cout << "x and y are NOT equal";
}
```

All **if** and **if-else** code can be written this way so that curly braces (also called *set braces*) always appear even when the statement blocks include just one statement each. This is extra work, but many programmers recommend it. The advantage of this approach is that it is clearer and easier to maintain, especially when you go back and add new statements.

Consider a simple **if-else** statement:

```
if (x == y)
    cout << "x and y equal";
else
    cout << "x, y not equal";
```

The following diagram illustrates the flow of control:



Interlude: Why Two Operators (= and ==)?

If you've used another programming language such as Pascal or Basic, you may wonder why = and == are two separate operators. After all, Basic uses a single equal sign (=) for both assignment and test-for-equality, using context to tell them apart.

In C and C++, the following code is freely permitted. Yet it's almost always wrong.

```
int x, y;
...
if (x = y)                    // ERROR! Assignment!
    cout << "x and y are equal";
```

What this example does is assign the value of y to x and use that value as the test condition. If this value is nonzero, it is considered "true." Consequently, if y is any value other than zero, the previous condition is "true," and the statement is always executed!

Here is the correct version, which will do what you want:

```
if (x == y)          // CORRECT: test for equality
    cout << "x and y are equal";
```

Here, "x == y" is an operation that tests for equality and evaluates as true or false. The important thing to remember is not to confuse test-for-equality with assignment (x = y).

Why allow this potential source of problems? Well, most expressions in C++ (the main exception being calls to **void** functions) produce a value, and this includes assignment

(=), which is an expression with a side effect. So, you can initialize three variables at once by doing this:

```
x = y = z = 0;   // Set all vars to 0.
```

which is equivalent to this:

```
x = (y = (z = 0));   // Set all vars to 0.
```

Each assignment, beginning with the rightmost one (z = 0), produces the value that was assigned (0), which is then used in the next assignment (y = 0). In other words, 0 is passed along three times, each time to a new variable.

C++ treats "x = y" as an expression that produces a value. And there'd be nothing wrong with that, except that *almost any valid numeric expression can be used as a condition*. Therefore, the compiler does not complain if you write this, even though it's usually wrong:

```
if (x = y)
    // Do action... (Probably should have used ==)
```