The *condition* is a true-or-false comparison that `while` examines — the same type of deal you find in an `if` comparison. If the *condition* is true, the *statements* in the loop are repeated. They keep doing so until the *condition* is false, and then the program continues. But, no matter what, the statements are always gone through once.

An important thing to remember here is that the `while` at the end of the loop requires a semicolon. Mess it up and it's sheer torture later to figure out what went wrong.

One strange aspect of the `do-while` loop is that it seriously lacks the *starting, while-true,* and *do-this* aspects of the traditional `while` and `for` loops. It has no starting condition because the loop just dives right into it. Of course, this sentence doesn't mean that the loop would lack those three items. In fact, it may look like this:

```
starting;
do
{
      statement(s);
      do_this;
}
while(while_true);
```

Yikes! Better stick with the basic `while` loop and bother with this jobbie only when something needs to be done once (or upside down).

- The condition that `while` examines is either TRUE or FALSE, according to the laws of C, the same as a comparison made by an `if` statement. You can use the same symbols used in an `if` comparison, and even use the logical doodads (`&&` or `||`) as you see fit.
- This type of loop is really rare. It has been said that only a mere 5 percent of all loops in C are of the `do-while` variety.
- You can still use `break` to halt a `do-while` loop. Only by using `break`, in fact, can you halt the statements in the midst of the loop. Otherwise, as with a `while` or `for` loop, all the statements within the curly braces repeat as a single block.

A flaw in the COUNTDWN.C program

Run the COUNTDWN program again. When it asks you to type a number, enter **200**.

There isn't a problem with this task; the program counts down from 200 to 0 and blasts off as normal. But, 200 is out of the range the program asks you to type.