Previously we saw arithmetic operators (+, -, *, etc.) now let's look at some others.

**Boolean operators** are those that are used in comparing two values and return “true” or “false”:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equals</td>
</tr>
<tr>
<td>!=</td>
<td>not equals</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
</tbody>
</table>

Keep in mind that, unlike in other languages like Java, C does not have any boolean datatype. So these operators return 1 to indicate true, and 0 to indicate false.

For example:
```c
int x = 5, y = 11;
int a = x > y;
int b = x != y;
```

In this case, `a` would be equal to 0 because `x > y` is false. But `b` would be 1 because `x != y` is true. Note that the last two lines may look a little strange to some programmers; it may be better to do this:
```c
int x = 5, y = 11;
int a = (x > y);
int b = (x != y);
```

**Logical operators** are those that are used to perform the logical operations we saw at the beginning of the course, specifically AND and OR:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For OR, the symbols are “pipes”, which is the key above the Enter key on most keyboards.

So we could have something like this:
```c
int x = 5, y = 11;
int a = (x == y) || (y > 10);
```

Here, `a` would be 1 because although `x == y` is false, `y > 10` is true, and with logical OR we only need one to be true.
**Conditionals**

Although it's certainly possible to use variables to hold the results of boolean and logical operators, usually those operators are used to determine whether to execute some code. That is, some code may be conditionally executed, depending on some boolean or logical operation.

The simplest structure for doing this in C is to use an **if-statement**. For instance, here's how we would read a variable and then, if it's negative, convert it to its (positive) absolute value:

```c
int x = 0;
scanf("%d", &x);
if (x < 0) {
    x = x * -1;
}
```

In this case, the line `x = x * -1` is only executed if `x < 0`, but is not executed otherwise. Note that there is no semicolon directly after the if-statement, just the open curly brace.

We can use an **if-else-statement** if we want to do something if a condition is true, and something else if it is false:

```c
int x = 0;
scanf("%d", &x);
if (x % 2 == 0) {
    printf("%d is even\n", x);
} else {
    printf("%d is odd\n", x);
}
```

Last, we can chain together if-else-statements to look for various conditions and do something if one of them is found to be true:

```c
int first = 0, second = 0;
scanf("%d", &first);
scanf("%d", &second);
if (first > second) {
    printf("The first number is bigger.\n");
} else if (second > first) {
    printf("The second number is bigger.\n");
} else {
    printf("The numbers are equal.\n");
}
```
BE CAREFUL! One of the most common programming errors in C is to use single-equals ("=", or assignment) when you mean to use double-equals ("==", or equal-to). Let's say you want to print a statement if the variable x is equal to 4:

```c
if (x == 4) {
    printf("hello!\n");
}
```

but you accidentally wrote this:

```c
if (x = 4) {
    printf("hello!\n");
}
```

In this case, the program always prints “hello!” Why? Because the assignment `x = 4` will evaluate to 4 (the value that is assigned to `x`), and any number not equal to 0 is considered “true”.

As you can imagine, if you accidentally did this:

```c
if (x = 0) {
    printf("hello!\n");
}
```

the program would never print “hello!” because the assignment `x = 0` will evaluate to 0, which is considered “false”.

In languages like Java, you cannot use the single-equals/assignment in an if-statement, but in C it's legal. So be careful!

**Loops**

With conditional statements, we either execute the code zero or one times. What if we wanted to execute it lots of times? In that case, we would use a loop.

One type of loop is the while-loop. It keeps executing while some condition is true. Here's an example in which we're trying to determine whether a variable named value is prime:

```c
int factor = 2;
while (value % factor != 0) {
    factor++;
}
if (value == factor) {
    printf("prime\n");
} else {
    printf("not prime; %d is a factor\n", factor);
}
```
Here, the “body” of the loop (which is just the statement `factor++`) continuously executes while the condition `value % factor != 0` is true. If and when the condition becomes false, then the loop terminates and we go on to the next line.

Be careful of infinite loops! That is, loops for which the condition is never false.

A variation of the while-loop is the **do-while-loop**. It also keeps looping while the condition is true, but the main difference is that the body of the loop is guaranteed to execute at least once:

```c
int factor = 1;
do {
    factor++;
} while (value % factor != 0);
```

Don't forget the semi-colon after the condition!! Note that we need to start with `factor` equal to 1, since we know it will necessarily be incremented before we check the condition.

The last type of loop is a **for-loop**. It typically looks something like this:

```c
int i;
for (i = 0; i < 10; i++) {
    printf("i is %d\n", i);
}
```

This loop will print “i is 0”, then “i is 1”, and so on up until “i is 9” but not “i is 10”. Let's see why:
- the first thing that happens is that the statement `i = 0` is executed. This only happens once, at the beginning of the for-loop.
- then, the condition `i < 10` is checked. If it's false, we break out of the loop. If it's true (which it is), then we proceed into the body of the loop and print “i is 0”.
- then, we execute the statement `i++`, which increments `i` so that it's now 1.
- we then go back to check the condition and proceed from there.

Note that after it prints “i is 9”, `i` gets incremented to 10, and then the condition `i < 10` is no longer true; that's why the loop ends after printing “i is 9”. Note, though, that the loop did get executed 10 times.

One thing to note about the for-loop is that it's typical to use `i` as the variable name (often referred to as the “loop index variable”). That's pretty much the only time when using a single-letter variable name is acceptable.

Also note that, in other languages like Java, for-loops usually look like this:

```java
for (int i = 0; i < 10; i++) {
    printf("i is %d\n", i);
}
```

The difference being that the loop index variable `i` is declared inside the for-loop header, not outside.
Some C compilers will allow this; some won't. So it's safer to declare `i` outside the loop.

One last thing: a for-loop can be written as a while-loop, and vice versa. All we need to do is move around the different statements and the condition check. For instance, the for-loop that prints “i is 0”, “i is 1”, etc. can be written as a while-loop like this:

```c
int i = 0;
while (i < 10) {
    printf("i is %d\n", i);
    i++;
}
```