Conditionals

Introduction

The C language has *conditional* statements, also called *selection* statements. Essentially, depending on a certain *condition*, a program can decide which statements to execute and which ones to ignore.

The simplest selection statement is the **`if`** statement:

```
if ( expression )
  statement
```

Note that the parentheses after the **`if`** keyword are required.

You read this as:

> *"If **expression** is true, then execute **statement**."*

You could also read it as:

> *"If **expression** is false, then do not execute **statement**."* (In which case **statement** is simply skipped.)

Notes about *expression*:

- *expression* is a *boolean* expression, meaning it is either **true** or **false**.
- Since there is no boolean type in C, zero evaluates to false and non-zero evaluates to true.
- To determine the value of the expression, you simply evaluate it.
- Assuming a is 5 and b is 0, these expressions are all **true**:

  ```
  a > b     a     a > 2     2 < a     b < 2     a - b     2     a * 5 * b + 4
  ```

- Assuming a is 5 and b is 0, these expressions are all **false**:

  ```
  b > a     b     a < 2     2 > a     a - 5     0          a * 5 * b
  ```

Relational operators:

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Equality operators:

Operator	Meaning
==	equal to
!=	not equal to

Note that the relational operators have higher precedence than the equality operators. (Operators in C)

Some example usage:

Statement	Correct/Incorrect

`if (a > 5)`
`  statement`
	Correct

`if (a)`
`  statement`
	Correct

`if (1)`
`  statement`
	Correct

`if a < 5`
`  statement`
	Missing parentheses

IF (a < 5) statement	Wrong 'if' keyword
if (a < 5) **then** statement	No 'then' keyword

`if ()`
`  statement`
	Missing expression

The value of a relational expression is either 0 (false) or 1 (true).

Examples of the relationship between false/0 and true/1:

```
int a = 5;
int b = 0;

printf("Value of a > b is %i\n", a > b);
printf("Value of a < b is %i\n", a < b);
printf("Value of a == b is %i\n", a == b);
printf("Value of a == a is %i\n", a == a);
printf("Value of b == b is %i\n", b == b);
printf("Value of a != a is %i\n", a != a);
printf("Value of a > a is %i\n", a > a);
printf("Value of b > b is %i\n", b > b);
```

Output:

```
Value of a > b is 1
Value of a < b is 0
Value of a == b is 0
Value of a == a is 1
Value of b == b is 1
Value of a != a is 0
Value of a > a is 0
Value of b > b is 0
```

Logical operators: (the precedence is accurate as well)

Operator	Meaning
!	logical not (negation)
&&	logical and
||	logical or

Boolean Truth Tables:

a	b	a && b	a || b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

a	b	a && b	a || b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Notes about these operators:

- Make sure you pay attention to the precedence of the operators.
- All the expressions will evaluate to 0 or 1 (false or true).
- The logical operators perform *short circuit evaluation*, meaning, as soon as the result can be determined, the evaluation stops.
- In English, this means:
  - True **or** anything is *true*. (Short circuit: Won't bother evaluating *anything*)
  - False **and** anything is *false*. (Short circuit: Won't bother evaluating *anything*)
  - False **or** anything is *anything*. (Must evaluate *anything*)
  - True **and** anything is *anything*. (Must evaluate *anything*)

  For example, what is the output of this program?

```
#include <stdio.h>

int main(void)
{
  int a;
  int b;

  a = 5;
  b = 3;
  if (a > b && b > 0 && ++a == 6)
    printf("1. The value of a is %i\n", a);

  a = 5;
  if (a > b && b > 5 && ++a == 6)
    printf("2. The value of a is %i\n", a);

  a = 5;
  if (a > b || b > 5 || ++a == 6)
    printf("3. The value of a is %i\n", a);

  a = 5;
  if (a > b && b > 5 || ++a == 6)
    printf("4. The value of a is %i\n", a);

  return 0;
}
```

- Also, don't be afraid to use redundant parentheses and spaces to help you understand:

```
if ( ( (a > b) && (b > 0) ) || (++a == 6) )
  printf("1. The value of a is %i\n", a);
```

  In fact, the GNU compiler will actually warn you about the lack of parentheses to get you to make your intentions clearer. (When mixing `||` and `&&`)

Note: Remember, the logical operators, `||` and `&&` are different from the other operators we've seen. These operators enable *short-circuit* evaluation so it is possible that a portion of the expression could be skipped entirely. This means that if there are any side-effect operators in the part of the expression that is skipped, those side-effects will **NOT** occur.

More on the `if` Statement

We've seen the simplest form of the **`if`** statement:

```
if ( expression )
  statement
```

where *statement* is exactly one statement. If you want to execute multiple statements, you need to include curly braces around them:

```
if ( expression )
{
  statements
}
```

The *statements* (plural) means more than one statement. Example:

```
  /* single statement */
if (a > b)
  printf("a = %i, b = %i\n", a, b);

  /* compound statement */
if (a > b)
{
  printf("a = %i, ", a);
  printf("b = %i\n", b);
}
```

Note that there is no semicolon after the closing curly brace. (But each statement inside the braces ends with a semicolon.) Also, it doesn't hurt to put a single statement inside curly braces:

```
  /* Braces unnecessary, but fine. */
if (a > b)
{
  printf("a = %i, b = %i\n", a, b);
}
```

This is also legal:

```
  /* Pointless, but fine. */
if (a > b)
```

```
{
}
```

However, without the braces, you can't have an empty statement. You'll need at least a semicolon:

```
  /* Pointless again, but legal. */
if (a > b)
    ;
```

Watch out for this common beginner's error which claims that 0 is greater than 5:

```
int a = 5;
int b = 0;

if (b > a);
  printf("b is greater than a\n");
```

The `else` Clause

Another form of the **`if`** statement includes an optional **`else`** clause:

```
if ( expression )
  statement1
else
  statement2
```

This reads as: "If *expression* is true, execute *statement*$_1$, otherwise, execute *statement*$_2$. This is *mutually exclusive*. Either *statement*$_1$ or *statement*$_2$ will get executed, but not both (or neither).

Either of the statements (or both) can be compound as well:

```
if ( expression )
{
  statements
}
else
  statement
```

```
if ( expression )
  statement
else
{
  statements
}
```

```
if ( expression )
{
  statements1
}
else
{
  statements2
}
```

Example:

```
int average = 85;
char grade;

if (average >= 70)
{
  grade = 'P';
  printf("You passed. Your average is %i%%.\n", average);
}
else
{
  grade = 'F';
  printf("You didn't pass. Your average is %i%%\n", average);
}
```

Nested `if` Statements

Sometimes we need to perform more than one test to determine the path our program will take. If the conditionals are mutually exclusive, we can *cascade* or *nest* the `if` statements.

Examples:

Non-nested

```
if (average >= 90)
  grade = 'A';
if (average >= 80)
  grade = 'B';
if (average >= 70)
  grade = 'C';
if (average >= 60)
  grade = 'D';
if (average < 60)
  grade = 'F';
```

Nested (cascading)

```
if (average >= 90)
  grade = 'A';
else
  if (average >= 80)
    grade = 'B';
  else
    if (average >= 70)
      grade = 'C';
    else
      if (average >= 60)
        grade = 'D';
      else
        grade = 'F';
```

Nested (no formatting)

```
if (average >= 90)
  grade = 'A';
else
if (average >= 80)
  grade = 'B';
else
if (average >= 70)
  grade = 'C';
else
if (average >= 60)
  grade = 'D';
else
  grade = 'F';
```

Can you see why the non-nested version will possibly execute slower than the nested version (besides being incorrect)?

The proper way to format nested **`if`** statements in this class:

```
if (average >= 90)
  grade = 'A';
else if (average >= 80)
  grade = 'B';
else if (average >= 70)
  grade = 'C';
else if (average >= 60)
  grade = 'D';
else
  grade = 'F';
```

Remember that the compiler doesn't care about formatting and will actually see this, all on one line:

```
if (average >= 90) grade = 'A'; else if (average >= 80) grade = 'B'; else if (average >= 70) grade = 'C'; else if (average >= 60) grade = 'D'; else grade = 'F';
```

In fact, can you take out all of the spaces as well? If not, which ones can you take out?

The "Dangling" `else`

This doesn't print out what you might expect:

```
if (average < 90)
  if (average < 60)
    printf("Failing\n");
else
  printf("An A student!\n");
```

If we change the formatting, we can see the problem more clearly.

```
if (average < 90)
  if (average < 60)
    printf("Failing\n");
  else
    printf("An A student!\n");
```

Again, compilers don't need any formatting, but humans do.

```
if (average < 90)
{
  if (average < 60)
    printf("Failing\n");
}
else
  printf("An A student!\n");
```

The rule for matching up `if` and `else` is:

> *The* `else` *matches the closest (previous)* `if` *that hasn't already been matched.*

You override this behavior through the use of braces, as shown above.

The Conditional Operator

Because the whole `if ... else . . .` idea is very common, C has created yet-another operator for this common situation.

expression$_1$ **?** *expression*$_2$ **:** *expression*$_3$

This reads as:

> If *expression*$_1$ is true, then execute *expression*$_2$, otherwise, execute *expression*$_3$.

There are two individual tokens: `?` and `:`, that are surrounded by expressions. Using this operator is pretty much same thing as we had with the `if ... else . . .`. In fact, each can be written in terms of the other. This:

```
if (a > b)
  printf("a is larger\n");
else
  printf("a is NOT larger\n");
```

is the same as this:

```
(a > b) ? printf("a is larger\n") : printf("a is NOT larger\n");
```

Note that the parentheses around `a + b` are redundant, but help to make the expressions clearer.

These examples are the same and both assign the larger value to `c`:

```
  /* Assign larger value to c */                /* Assign larger value to c */
if (a > b)                                    c = (a > b) ? a : b;
  c = a;
else
  c = b;
```

Example:

```
int a = 1;
int b = 4;
int c;

  /* These two statements are the same */
c = a > b ? a + 2 : b + 2;
c = (a > b) ? (a + 2) : (b + 2);
```

What is printed from these statements?

```
c = a == b ? a + 2 : b + 2;
printf("a = %i, b = %i, c = %i\n", a, b, c);

c = a = b ? a + 2 : b + 2;
printf("a = %i, b = %i, c = %i\n", a, b, c);
```

What about these?

```
c = a = b ? a + 2 : b += 2;
c = (a = b) ? (a + 2) : (b += 2);
```

The `switch` Statement

The `switch` statement is similar to nested `if ... else ...` statements. The most common form of the `switch` statement looks like this:

```
switch ( expression )
{
  case constant_expression_1 :
    statements_1
    break;
  case constant_expression_2 :
    statements_2
    break;
  . . .
  case constant_expression_N :
    statements_N
    break;
}
```

An example showing both a nested `if ... else ...` statement and a `switch` statement. The result is the same. However, when you have a larger number of conditions, the `switch` statement may execute faster.

Nested `if`	`switch`

```
if (year == 1)
  printf("Freshman\n");
else if (year == 2)
  printf("Sophomore\n");
else if (year == 3)
  printf("Junior\n");
else if (year == 4)
  printf("Senior\n");
```

```
switch (year)
{
  case 1:
    printf("Freshman\n");
    break;
  case 2:
    printf("Sophomore\n");
    break;
  case 3:
    printf("Junior\n");
    break;
  case 4:
    printf("Senior\n");
    break;
}
```

Notice that if the value of `year` is not one of the values tested, nothing will be printed. If you want a *catch-all* condition, you would use an `else` clause in the `if` statement and for the `switch` statement, use a `default`:

Nested `if`

```
if (year == 1)
  printf("Freshman\n");
else if (year == 2)
  printf("Sophomore\n");
else if (year == 3)
  printf("Junior\n");
else if (year == 4)
  printf("Senior\n");
else
  printf("Invalid year\n");
```

`switch`

```
switch (year)
{
  case 1:
    printf("Freshman\n");
    break;
  case 2:
    printf("Sophomore\n");
    break;
  case 3:
    printf("Junior\n");
    break;
  case 4:
    printf("Senior\n");
    break;
  default:
    printf("Invalid year\n");
    break;
}
```

Notes:

- The `switch` (controlling) expression must be an *integral* type. (No `float`, `double`, etc.)
- Each `case` label must be an integral *constant expression*. The value must be known at compile time.
- No braces are required when you have multiple statements after a case label.
- The `break` statement causes execution to continue with the statement immediately after the entire `switch` statement.
- If a `case` does not contain a `break` statement, the next `case` is executed. (Fall-through) Example:

case

```
switch (year)
{
  case 1:
  case 2:
    printf("Lower division\n");
    break;
  case 3:
  case 4:
    printf("Upper division\n");
    break;
  default:
    printf("Invalid year\n");
    break;
}
```

if

```
if ( (year == 1) || (year == 2) )
  printf("Lower division\n");
else if ( (year == 3) || (year == 4) )
  printf("Upper division\n");
else
  printf("Invalid year\n");
```

Boolean Types

As was stated before, C doesn't have a boolean type. Instead, it uses 0 to represent false and 1 (or non-zero) to represent true.

Using one and zero, the meaning isn't clear:

```
int value = 1;

if (value == 1)
{
  /* do something if value is true */
}

if (value == 0)
{
  /* do something if value is false */
}
```

We can "create" our own boolean values and type:

```
#define FALSE 0
#define TRUE  1
#define BOOL int
```

And use these types in our programs:

Explicit comparisons

```
BOOL value = TRUE;

if (value == TRUE)
{
  /* do something if value is true */
}

if (value == FALSE)
{
  /* do something else if value is false */
}
```

Implicit comparison

```
BOOL value = TRUE;

if (value)
{
  /* do something if value is true */
}

if (!value)
{
  /* do something else if value is false */
}
```