C Programs: Boolean Expressions, Conditional Statements

Boolean Expressions (Comparisons, Conditional Expressions)

Truth and Falsehood

C does not have a specially designated data type to represent truth and falsehood values. (in other programming languages this is known as boolean type).

Instead, C uses its `int` type to represent truth and falsehood. The values of `int` type map to boolean (truth) values as follows:

Truth value	`int` value
false	0
true	any **non-zero** `int`

Comparisons

When two values are compared, the result of the comparison is a truth value (is it true that X is greater than Y? yes, it is true... no, it is false...).

In C, comparison of two values is an expression. C uses relational and equality operators to construct comparison expressions. The following list of relational/equality operators is available in C for comparison of `int`s and `float`s (as well as `char`s):

Operator	Meaning	True expression	False expression
$<$	less than	`3 < 5` evaluates to `1`	`5 < 3` evaluates to `0`
$>$	greater than	`5 > 3` evaluates to `1`	`3 < 5` evaluates to `0`
$<=$	less than or equal to	`5 <= 5` evaluates to `1`	`5 <= 3` evaluates to `0`
$>=$	greater than	`5 >= 3` evaluates to `1`	`3 >= 5` evaluates to `0`
$==$	equal to	`5 == 2+3` evaluates to `1`	`5 == 2+2` evaluates to `0`
$!=$	not equal to	`5 != 2+2` evaluates to `1`	`5 != 2+3` evaluates to `0`

Comparisons can be used in the same way as arithmetic expressions are used, as part of assignment statements. Note, true comparisons evaluate to 1, false comparisons evaluate to 0.

Example. Consider the following C program:

```
#include <stdio.h>
int main() {
 int x,y,b,c;
 x = 10;
 y = 5;
 b = (x == y);
 c = (x >= 2*y);
 printf("%d\n%d\n%d\n",b,c,(b!=c));
 return 0;
}
```

The output of the program is

```
0
1
1
```

Boolean (Logical) Operators

Often, one needs to obtain a truth value of a group of comparisons. This is done using three boolean or logical operations: conjunction, disjunction and negation:

conjunction computes the truth value of the statement "**all** comparisons in the group evaluate to true".

disjunction computes the truth value of the statement "**at least one** comparison in the group evaluates to true".

negation "flips" the truth value of a comparison.

Logical Operator	C operator	Arity
conjunction	&&	binary
disjunction	||	binary
negation	!	unary

The syntax of a logical operation expression is:

```
<Expression> <Logical Operator> <Expression>
```

where `<Logical Operator>` is either `&&` or `||`, or

```
! <Expression>
```

(conventions about the standard use of partentheses apply)

Examples. The following are correct C logical expressions:

```
(2 == 1+1) && (3 == 4-1)
1 || 0
!(2 == 100 -90)
(! (3 == 2) && (1 || 0))
```

Meaning of boolean operators. The formal semantics of the boolean operators is described in the following tables:

Semantics of conjunction (&&)

<Expression1>	<Expression2>	<Expression1> && <Expression2>
true (nonzero)	true (nonzero)	true (1)
true (nonzero)	false (0)	false (0)
false (0)	true (nonzero)	false (0)
false (0)	false (0)	false (0)

Semantics of disjunction (||)

<Expression1>	<Expression2>	<Expression1> || <Expression2>
true (nonzero)	true (nonzero)	true (1)
true (nonzero)	false (0)	true (1)
false (0)	true (nonzero)	true (1)
false (0)	false (0)	false (0)

Semantics of negation (!)

<Expression>	!<Expression>
true (nonzero)	false (0)
false (0)	true (1)

Operator precedence. The following is the operator precedence table for all known to us C operators (arithmetic, assignment, comparison, logical):

Precedence Level	Operators	Arity
1. (highest)	*function calls*	*varies*
2.	!, +, -	unary
3.	*, /, %	binary
4.	+,-	binary
5.	$<, <=, >=, >$	binary
6.	==, !=	binary
7.	&&	binary
8.	||	binary
9. (lowest)	=	binary

DeMorgan's Theorem (Law). *Behold the awesome power of logic!*

!(<Expr1> && <Expr2>) is (!<Expr1> || !<Expr2>)

!(<Expr1> || <Expr2>) is (!<Expr1> && !<Expr2>)

If statement

Comparisons and logical operations allow us to construct expressions whose values are interpreted as true and false (*boolean* expressions).

if statements (as well as switch statements to be discussed later) use *boolean* expressions to determine which statements need to be executed next.

Up until now, in **all** programs we considered, **all** statements were executed *exactly in the order they appeared in the text of the program.* Conditional C statements (if statement and switch statement) diverge from this.

If statement with one alternative. The syntax of the most simple if statement is

```
if ( <expression>)
   <statement-block>
```

Here, <expression> is any expression which evaluates to a truth value. <statement-block> is either a single statement or a sequence of statements enclosed in curly braces ("{", "}").

Examples. A single-alternative if statement with a single statement inside it:

```
if (x <= 5)
   y = x*x;
```

A simple single-alternative if statement with a statement block inside it:

```
if ( (x >= 5) && (x <= 20)) {
   y = x*x;
   x = x+1;
   printf("%d\n", y);
}
```

Note: A closing curly brace "}" is an indicator of the end of a group of statements. As such, **there is no need** to follow it with a semicolumn.

Semantics. Single-alternative `if` statement is evaluated as follows:

1. `<expression>` is evaluated.
2. If `<expression>` evaluates to true (i.e., to a non-zero integer), then the statement(s) from the `<statement-block>` is/are executed. After the last statement in `<statement-block>` is executed (unless it was a `return` statement), the statement immediately following the `if` statement will be executed.
3. If `<expression>` evaluated to false (i.e., 0), the statement immediately following the `if` statement will be executed.

If statement with two alternatives. A more complex version of the `if` statement has the following syntax:

```
if (<expression>)
   <true-statement-block>
else
   <false-statement-block>
```

Here, `<expression>` is any expression which evaluates to a truth value. `<true-statement-block>` and `<false-statement-block>` are either single statements or sequences of statements enclosed in curly braces ("{", "}").

Examples. A two-alternative `if` statement with a single statements inside it:

```
if (x <= y)
   printf("Max = %d\n",y);
else
   printf("Max = %d\n",x);
```

A simple single-alternative `if` statement with a statement block inside it:

```
if ( (x >= 5) && (x <= 20)) {
   y = x*x;
   x = x+1;
   printf("%d\n", y);
}
else {
   y = (x-1)*(x-1);
   x = x-1;
   printf("%d\n",x);
}
```

Semantics. Two-alternative `if` statement is evaluated as follows:

1. `<expression>` is evaluated.
2. If `<expression>` evaluates to true (i.e., to a non-zero integer), then the statement(s) from the `<true-statement-block>` is/are executed. After the last statement in `<false-statement-block>` is executed (unless it was a `return` statement), the statement immediately following the `if` statement will be executed.

3. If <expression> evaluated to false (i.e., 0), then he statement(s) from the <false-statement-block> is/are executed. After the last statement in <false-statement-block> is executed (unless it was a return statement), the statement immediately following the if statement will be executed.

Example. In the first example above, if y is greater than or equal to x, the if part of the statement will be executed, and the value of y will be printed to standard output. If y is not greater than or equal to x (i.e., if x is greater than y), the value of x will be printed to standard output.

Nested If statements

if statements can be parts of blocks of statement blocks inside other if statements. For example, the if ((x >= 5) && (x <= 20)) condition can be rewritten in a form nested if statements:

```
begin{verbatim}
 if (x >= 5) {
    if (x <= 20) {
       y = x*x;
       x = x+1;
       printf("%d\n", y);
    }
 }
```

Note: please note the indentation of the if statements. It is **extremely important** to keep proper track of the open and closed curly braces in your if statements, especially, if you are using nested ifs.