In C, assignments are expressions, not statements.

Embedded assignments - legal anywhere an expression is legal

- This allows multiple assignment `a = b = c = 1;`

Assignment operators

- Same precedence; right to left
- `= assignment`
- Perform the indicated operation between the left and right operands, then assign the result to the left operand
  - `+=` add to
  - `-=` subtract from
  - `*=` multiply by
  - `/=` divide by
  - `%=` modulo by

- Example 1: `a=x=y+3;` so `a = x`, right?
- Example 2: `r=s+(t=u-v)/3;` give “same as” code.

NOTE: using an assignment operator (=) is legal anywhere it is legal to compare for equality (==), so it is not a syntax error (depends on compiler; may give a warning) because there is not a distinct boolean type in C.
Boolean Operators

- C does not have a distinct boolean type
  - int is used instead

- Treats integer 0 as FALSE and all non-zero values as TRUE
  - i = 0;
  - while (i - 10) {
    - ... }
  - will execute until the variable i takes on the value 10 at which time the expression (i - 10) will become false (i.e. 0).

- A sampling of Logical/Boolean Operators:
  - &&, ||, and ! ➞ AND, OR, and NOT

- For example, && is used to compare two objects
  - x != 0 && y != 0

- Short-Circuit Evaluation: In the above example, if x != 0 evaluates to false, the whole statement is false regardless of the outcome of y != 0
  - same for OR if first condition is true
The boolean operators function in a similar way to the comparison (relational) operators: each returns 0 if evaluates to FALSE or 1 if it evaluates to TRUE.

**NOT**: The NOT operator accepts one input. If that input is TRUE, it returns FALSE, and if that input is FALSE, it returns TRUE. For example, NOT (1) evaluates to 0, and NOT (0) evaluates to 1. NOT (any number but zero) evaluates to 0. In C, NOT is written as !

**AND**: This is another important command. AND returns TRUE if both inputs are TRUE (if 'this' AND 'that' are true). (1) AND (0) would evaluate to zero because one of the inputs is false (both must be TRUE for it to evaluate to TRUE). (1) AND (1) evaluates to 1. (any number but 0) AND (0) evaluates to 0. The AND operator is written && in C. Do not be confused by thinking it checks equality between numbers: it does not.

**OR**: Very useful is the OR statement! If either (or both) of the two values it checks are TRUE then it returns TRUE. For example, (1) OR (0) evaluates to 1. (0) OR (0) evaluates to 0. The OR is written as || in C. Those are the pipe characters. On your keyboard, they may look like a stretched colon. On my computer the pipe shares its key with \. A “unary” operator since it only works on one operand.

**PRECEDENCE**
- NOT is evaluated prior to both AND and OR. Also multiple NOTs are evaluated from right to left.
- AND operator is evaluated before the OR operator. Also, multiple ANDs are evaluated from left to right.
- OR will be evaluated after AND. Also, multiple ORs are evaluated from left to right.

- **AND (&&)**: Returns true only if both operand are true.
- **OR (||)**: Returns true if one of the operand is true.
- **NOT (!)**: Converts false to true and true to false.
Boolean Examples

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator's Name</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
<td>3&gt;2 &amp;&amp; 3&gt;1</td>
<td>1 (true)</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
<td>3&gt;2 &amp;&amp; 3&lt;1</td>
<td>0 (false)</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
<td>3&lt;2 &amp;&amp; 3&lt;1</td>
<td>0 (false)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OR</td>
</tr>
<tr>
<td>!</td>
<td>NOT</td>
<td>!(3==2)</td>
<td>1 (true)</td>
</tr>
<tr>
<td>!</td>
<td>NOT</td>
<td>!(3==3)</td>
<td>0 (false)</td>
</tr>
</tbody>
</table>

A.  !(1 || 0)  ANSWER: 0
B.  !(1 || 1 && 0)  ANSWER: 0 (AND is evaluated before OR)
C.  !((1 || 0) && 0)  ANSWER: 1 (Parenthesis are useful)
Loop constructs

- Statement(s) only execute when the condition is TRUE.
- Notice the semi-colon locations.
- Do/while always executes at least once (not necessarily true for the other two constructs).
- For loop → initialize a value, test the condition, if condition is true, execute the statements (if condition is false, exit the loop), modify the value, test the condition, if the condition is true, execute the statements, etc.

```c
#include <stdio.h>

int main () {
    int n, sum;
    sum = 0;
    for (n = 1; n <= 10; n=n+1)
        { sum = sum + n;
    }
    printf("\n The sum of integers from 1 to 10 is %d, have a nice day", sum);
    return 0;
}

n=1;
while (n<=10) {
    sum=sum+n;
    n=n+1;
}

n=0;
do {
    n=n+1;
    sum=sum+n;
} while (n<10);
```
#include <stdio.h>
int main()
{
    int x;
    x = 0;
    do {
        printf( "Hello, world!\n" );
    } while ( x != 0 );
    getchar();
    return 0;
}

/* "Hello, world!" is printed at least one time even though the condition is false */

#include <stdio.h>
int main()
{
    int x = 0;
    while ( x < 10 ) {
        printf( "%d\n", x );
        x++;  }  /* Update x so the condition can be met eventually */
    getchar();
    return 0;  }

#include <stdio.h>
int main()
{
    int x;
    for ( x = 0; x < 10; x++ ) {
        printf( "%d\n", x );
    }
    getchar();
    return 0;  }

/* The loop goes while x < 10, and x increases by one every loop*/
Keywords that are very important to looping are **break** and **continue**.

**BREAK** command will
- exit the most immediately surrounding loop regardless of what the conditions of the loop are.
- Break is useful if we want to exit a loop under special circumstances.

**CONTINUE** is another keyword that controls the flow of loops. If you are executing a loop and hit a continue statement, the loop will stop its current iteration, update itself (in the case of **FOR** loops) and begin to execute again from the top. Essentially, the continue statement is saying "this iteration of the loop is done, let's continue with the loop without executing whatever code comes after me."

**GOTO** next slide (really, continued on the next slide)
Jump Statement

- Goto plus a labeled statement
  - `goto identifier;`
  - `identifier: statement;`

- Have to declare the identifier???
  - NO!

- A statement label is meaningful only to a `goto` statement; in any other context, a labeled statement is executed without regard to the label (i.e. is ignored).

- A `jump-statement` must reside in the same function and can appear before only one statement in the same function.
- The set of `identifier` names following a `goto` has its own name space so the names do not interfere with other identifiers.
- Labels cannot be redeclared.

- It is good programming style to use the `break`, `continue`, and `return` statement in preference to `goto` whenever possible. However, since the `break` statement only exits from one level of the loop, a `goto` may be necessary for exiting a loop from within a deeply nested loop.
while (true) {
    if (someone_has_won() || someone_wants_to_quit() == TRUE) {
        break;
    }
    take_turn(player1);
    if (someone_has_won() || someone_wants_to_quit() == TRUE) {
        break;
    }
    take_turn(player2); } /* checkers */

for (player = 1; someone_has_won == FALSE; player++) {
    if (player > total_number_of_players) {
        player = 1;
    }
    if (is_bankrupt(player)) {
        continue;
    }
    take_turn(player); } /* monopoly */
Goto example

#include <stdio.h>
int main() {
    int i, j;
    for ( i = 0; i < 10; i++ )
    {
        printf_s( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 3; j++ )
        {
            printf_s( " Inner loop executing. j = %d\n", j );
            if ( i == 5 )
                goto stop;
        }
    }

    /* This message does not print: */
    printf_s( "Loop exited. i = %d\n", i );
    stop: printf( "Jumped to stop. i = %d\n", i );
    return 0; }

In this example, a goto statement transfers control to the point labeled stop when i equals 5.