Boolean Data Type & Expressions Outline

1. Boolean Data Type & Expressions Outline
2. Basic Data Types
3. C Boolean Data Types: `char` or `int`/ Boolean Declaration
4. Boolean or Character?
5. Boolean Literal Constants
6. Boolean Expressions/ Boolean Operations
7. Example: Boolean Literal Constants & Expressions
8. Example: Boolean Variables
9. Relational Operations
10. Example: Relational Expressions
11. Structure of Boolean Expressions
12. Precedence Order of Boolean Operations
13. Another Boolean Precedence Example
14. Example: Boolean Precedence Order
15. Example: More Relational Expressions
16. Example: More Relational Expressions (continued)
17. Short Circuiting
18. Exercise: Writing a Program

Basic Data Types

- Numeric
  - `int`
  - `float`
- Non-numeric
  - `char`

```
#include <stdio.h>

int main ()
{ /* main */
    float standard_deviation, relative_humidity;
    int   count, number_of_silly_people;
    char  middle_initial, hometown[30];
} /* main */
```

C Boolean Data Type: `char` or `int`

The C data type typically used for storing Boolean values is `char`, although `int` will also work.

Like numeric data types, Booleans have particular ways of being stored in memory and of being operated on.

Conceptually, a Boolean value represents a single bit in memory, although the `char` and `int` data types aren't implemented this way — if for no other reason than that computers can't address a single bit, since the smallest collection of bits that they can address is a byte (or, in a few cases, a word).

Boolean Declaration

```
char CS1313_lectures_are_fascinating;
```

This declaration tells the compiler to grab a group of bytes, name them `CS1313_lectures_are_fascinating`, and think of them as storing a Boolean value (either True or False).

How many bytes?

Even though conceptually a Boolean represents a single bit, in practice `char` variables are usually implemented using 8 bits (1 byte):

`CS1313_lectures_are_fascinating` :

?	?	?	?	?	?	?	?

Boolean or Character?

Question: How does the C compiler know that a particular `char` declaration is a Boolean rather than a character?

Answer: It doesn't.

```
% cat shortcircuit.c
#include <stdio.h>

int main ()
{ /* main */
    const int maximum_short_height_in_cm = 170;
    int  my_height_in_cm = 160;
    char I_am_Henry = 1;
    char I_am_tall;
    char my_middle_initial = 'J';

    I_am_tall =
        (!I_am_Henry) ||
        (my_height_in_cm > maximum_short_height_in_cm);
    printf("I_am_Henry        = %d\n", I_am_Henry);
    printf("my_height_in_cm   = %d\n", my_height_in_cm);
    printf("I_am_tall         = %d\n", I_am_tall);
    printf("my_middle_initial = %c\n", my_middle_initial);
} /* main */

% gcc -o shortcircuit shortcircuit.c
% shortcircuit
I_am_Henry        = 1
my_height_in_cm   = 160
I_am_tall         = 0
my_middle_initial = J
```

Whether a `char` is a Boolean or a character depends **entirely** on how you use it in the program.

Boolean Literal Constants

A *Boolean literal constant* can have either of two possible values (but not both):

to represent false:	0
to represent true:	anything other than 0 (usually 1)

We can use Boolean literal constants in several ways:

- In declaring and initializing a **named constant**:
  ```
  const char true = 1;
  ```
- In declaring and initializing a **variable**:
  ```
  char getting_a_bad_grade = 0;
  ```
- In an **assignment**:
  ```
  this_is_my_first_guess = 1;
  ```
- In an **expression**:
  ```
  Henry_is_short && 1;
  ```

The first two of these uses are considered good programming practice, **AND SO IS THE THIRD**, which is a way that Booleans are different from numeric data; that is, it's acceptable to use a Boolean literal constant in an assignment statement.

As for using Boolean literal constants in expressions, it's not so much that it's considered bad programming practice, it's just that it's kind of pointless.

Boolean Expressions

Just like a numeric arithmetic expression, a *Boolean expression* is a combination of Boolean terms (such as variables, named constants and literal constants), Boolean operators (e.g., `!`, `&&`, `||`, relational comparisons) and parentheses.

```
I_am_happy
!I_am_happy
it_is_raining && it_is_cold
it_is_raining || it_is_cold
(!it_is_raining) || (it_is_cold && I_am_happy)
```

Boolean Operations

Like arithmetic operations, Boolean operations come in two varieties: *unary* and *binary*. A unary operation is an operation that uses only one term; a binary operation uses two terms. Boolean operations include:

Operation Name	Kind	Operator	Usage	Effect
Identity	Unary	None	`x`	None
Negation	Unary	`!`	`!x`	Inverts value of `x`
Conjunction (AND)	Binary	`&&`	`x && y`	1 if both `x` is nonzero and `y` is nonzero; otherwise 0
Disjunction (Inclusive OR)	Binary	`||`	`x || y`	1 if either `x` is nonzero or `y` is nonzero, or both; otherwise 0

Note: C Boolean expressions evaluate either to 0 (representing false) or to 1 (representing true).

Example:
Boolean Literal Constants & Expressions

```
% cat lgcexprsimple.c
#include <stdio.h>

int main ()
{ /* main */
    const char true = 1, false = 0;

    printf(" true = %d,  false = %d\n",  true,  false);
    printf("!true = %d, !false = %d\n", !true, !false);
    printf("\n");
    printf("true  || true  = %d\n", true  || true);
    printf("true  || false = %d\n", true  || false);
    printf("false || true  = %d\n", false || true);
    printf("false || false = %d\n", false || false);
    printf("\n");
    printf("true  && true  = %d\n", true  && true);
    printf("true  && false = %d\n", true  && false);
    printf("false && true  = %d\n", false && true);
    printf("false && false = %d\n", false && false);
} /* main */
% gcc -o lgcexprsimple lgcexprsimple.c
% lgcexprsimple
 true = 1,  false = 0
!true = 0, !false = 1

true  || true  = 1
true  || false = 1
false || true  = 1
false || false = 0

true  && true  = 1
true  && false = 0
false && true  = 0
false && false = 0
```

Notice that a Boolean expression always evaluates to either `1` or `0`.

Example: Boolean Variables

```
% cat prog_logic.c
#include <stdio.h>

int main ()
{ /* main */
    int project_due_soon;
    int been_putting_project_off;
    int start_working_on_project_today;

    printf("Is it true that you have a ");
    printf("programming project due soon?\n");
    scanf("%d", &project_due_soon);
    printf("Is it true that you have ");
    printf("been putting off working on it?\n");
    scanf("%d", &been_putting_project_off);
    start_working_on_project_today =
        project_due_soon && been_putting_project_off;
    printf("Is it true that you should start ");
    printf("working on it today?\n");
    printf("ANSWER:  %d\n",
        start_working_on_project_today);
} /* main */
% gcc -o prog_logic prog_logic.c
% prog_logic
Is it true that you have a programming project due soon?
 1
Is it true that you have been putting off working on it?
 1
Is it true that you should start working on it today?
ANSWER:  1
```

Relational Operations

A *relational* operation is a binary operation that compares two numeric operands and produces a Boolean result. For example:

```
CS1313_lab_section == 14
         cm_per_km != 100
               age <  21
number_of_students <= number_of_chairs
      credit_hours >  30
   electoral_votes >= 270
```

The relational operations are:

Operation Name	Operand	Usage	Result
Equal	`==`	`x == y`	1 if the value of `x` is exactly the same as the value of `y`; otherwise 0
Not Equal	`!=`	`x != y`	1 if the value of `x` is different from the value of `y`; otherwise 0
Less Than	`<`	`x < y`	1 if the value of `x` is less than the value of `y`; otherwise 0
Less Than Or Equal To	`<=`	`x <= y`	1 if the value of `x` is less than or equal to the value of `y`; otherwise 0
Greater Than	`>`	`x > y`	1 if the value of `x` is greater than the value of `y`; otherwise 0
Greater Than Or Equal To	`>=`	`x >= y`	1 if the value of `x` is greater than or equal to the value of `y`; otherwise 0

Example: Relational Expressions

```
% cat relational.c
#include <stdio.h>

int main ()
{ /* main */
    int CS1313_size, METR2413_size;

    printf("How many students are in CS1313?\n");
    scanf("%d", &CS1313_size);
    printf("How many students are in METR2413?\n");
    scanf("%d", &METR2413_size);
    printf("%d == %d: %d\n", CS1313_size, METR2413_size,
       CS1313_size == METR2413_size);
    printf("%d != %d: %d\n", CS1313_size, METR2413_size,
       CS1313_size != METR2413_size);
    printf("%d <  %d: %d\n", CS1313_size, METR2413_size,
       CS1313_size <  METR2413_size);
    printf("%d <= %d: %d\n", CS1313_size, METR2413_size,
       CS1313_size <= METR2413_size);
    printf("%d >  %d: %d\n", CS1313_size, METR2413_size,
       CS1313_size >  METR2413_size);
    printf("%d >= %d: %d\n", CS1313_size, METR2413_size,
       CS1313_size >= METR2413_size);
} /* main */
% gcc -o relational relational.c
% relational
How many students are in CS1313?
 64
How many students are in METR2413?
 57
64 == 57: 0
64 != 57: 1
64 <  57: 0
64 <= 57: 0
64 >  57: 1
64 >= 57: 1
```

Structure of Boolean Expressions

A Boolean expression can be long and complicated. For example:

```
a || b || c && d && !e
```

Terms and operators can be mixed together in almost limitless variety, but they must follow the rule that a unary operator has a term immediately to its right and a binary operator has terms on both its left and its right.

Parentheses can be placed around any unary or binary expression:

```
(a || b) || (c && (d && (!e)))
```

Putting a term in parentheses may change the value of the expression, because a term inside parentheses will be calculated first. For example:

```
a || b && c
```

is evaluated as “b AND c, OR a,” but

```
(a || b) && c
```

is evaluated as “a OR b, AND c.”

So, Boolean expressions look and behave a lot like arithmetic expressions.

Precedence Order of Boolean Operations

In the absence of parentheses to explicitly state the order of operations, the order of precedence is:

1. relational operations, left to right
2. `!`, left to right
3. `&&`, left to right
4. `||`, left to right

After taking into account the above rules, the expression as a whole is evaluated left to right. For example:

```
0 &&  1 || 1  && 1
  ⇓
  0     || 1  && 1
           ⇓
  0     ||    1
        ⇓
        1
```

but

```
0 && (1 || 1) && 1
        ⇓
0 &&    1     && 1
  ⇓
  0           && 1
              ⇓
              0
```

Another Boolean Precedence Example

```
!  0 || 1
 ⇓
 1   || 1
      ⇓
      1

      but

! (0 || 1)
      ⇓
!     1
   ⇓
   0
```

Rule of Thumb: if you can't remember the priority order of the operators, use lots of parentheses.

Example: Boolean Precedence Order

```
% cat lgcexpr.c
#include <stdio.h>

int main ()
{ /* main */
    printf("0 &&  1 || 1  && 1 = %d\n",
           0 &&  1 || 1  && 1);
    printf("0 && (1 || 1) && 1 = %d\n",
           0 && (1 || 1) && 1);
    printf("! 0 || 1  = %d\n",
           ! 0 || 1);
    printf("!(0 || 1) = %d\n",
           !(0 || 1));
} /* main */
% gcc -o lgcexpr lgcexpr.c
% lgcexpr
0 &&  1 || 1  && 1 = 1
0 && (1 || 1) && 1 = 0
! 0 || 1  = 1
!(0 || 1) = 0
```

Example: More Relational Expressions

```
% cat comparisons.c
#include <stdio.h>

int main ()
{ /* main */
    int  a, b, c;
    char b_equals_a, b_equals_c;
    char b_between_a_and_c, b_between_c_and_a;
    char b_outside_a_and_c;
    char a_lt_b_lt_c, c_lt_b_lt_a;

    printf("Enter three different integers:\n");
    scanf("%d %d %d", &a, &b, &c);
    printf("The integers you entered are:\n");
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    b_equals_a        = (b == a);
    b_equals_c        = (b == c);
    b_between_a_and_c = ((a < b) && (b < c));
    b_between_c_and_a = ((c < b) && (b < a));
    b_outside_a_and_c =
        !(b_between_a_and_c || b_between_c_and_a);
    a_lt_b_lt_c       = a < b < c;
    c_lt_b_lt_a       = c < b < a;
    printf("b == a:            %d\n", b_equals_a);
    printf("b == c:            %d\n", b_equals_c);
    printf("a < b && b < c:    %d\n", b_between_a_and_c);
    printf("c < b && b < a:    %d\n", b_between_c_and_a);
    printf("a < b < c:         %d\n", a_lt_b_lt_c);
    printf("c < b < a:         %d\n", c_lt_b_lt_a);
    printf("b outside a and c: %d\n", b_outside_a_and_c);
} /* main */
% gcc -o comparisons comparisons.c
% comparisons
Enter three different integers:
 4 4 5
The integers you entered are:
a = 4, b = 4, c = 5
b == a:            1
b == c:            0
a < b && b < c:    0
c < b && b < a:    0
a < b < c:         1
c < b < a:         1
b outside a and c: 1
% comparisons
Enter three different integers:
 4 5 5
The integers you entered are:
a = 4, b = 5, c = 5
b == a:            0
b == c:            1
a < b && b < c:    0
c < b && b < a:    0
```



```
a < b < c:         1
c < b < a:         1
b outside a and c: 1
% comparisons
Enter three different integers:
 4 5 6
The integers you entered are:
a = 4, b = 5, c = 6
b == a:            0
b == c:            0
a < b && b < c:    1
c < b && b < a:    0
a < b < c:         1
c < b < a:         1
b outside a and c: 0
% comparisons
Enter three different integers:
 6 5 4
The integers you entered are:
a = 6, b = 5, c = 4
b == a:            0
b == c:            0
a < b && b < c:    0
c < b && b < a:    1
a < b < c:         1
c < b < a:         1
b outside a and c: 0
% comparisons
Enter three different integers:
 4 3 5
The integers you entered are:
a = 4, b = 3, c = 5
b == a:            0
b == c:            0
a < b && b < c:    0
c < b && b < a:    0
a < b < c:         1
c < b < a:         1
b outside a and c: 1
```

Short Circuiting

When a C program evaluates a Boolean expression, it may happen that, after evaluating some of the terms, the result can no longer change:

```
% cat shortcircuit.c
#include <stdio.h>

int main ()
{ /* main */
    const int maximum_short_height_in_cm = 170;
    int  my_height_in_cm = 160;
    char I_am_Henry = 1;
    char I_am_tall;
    char my_middle_initial = 'J';

    I_am_tall =
        (!I_am_Henry) ||
        (my_height_in_cm > maximum_short_height_in_cm);
    printf("I_am_Henry        = %d\n", I_am_Henry);
    printf("my_height_in_cm   = %d\n", my_height_in_cm);
    printf("I_am_tall         = %d\n", I_am_tall);
    printf("my_middle_initial = %c\n", my_middle_initial);
} /* main */

% gcc -o shortcircuit shortcircuit.c
% shortcircuit
I_am_Henry        = 1
my_height_in_cm   = 160
I_am_tall         = 0
my_middle_initial = J
```

In such a case, the Boolean expression *short circuits*: the rest of the expression is not evaluated, because evaluating it takes time, but will not change the result.

In the example above, the relational expression never gets evaluated, because the first operand in the OR operation (`||`) evaluates to 1, and therefore the entire OR operation must evaluate to 1.

Exercise: Writing a Program

Let *k* be a distance in kilometers. Let *m* be a distance in millimeters. Then:

$$k = m \cdot 10^6$$

Write a C program that inputs an `int` distance in kilometers and an `int` distance in millimeters, and determines whether they are the same distance.

The body of the program must not have any numeric literal constants; all constants must be declared using appropriate variable names.

Don't worry about comments.