Boolean Data Outline

Data Types

A **data type** is (surprise) a type of data:

- Numeric
  - `int`: ***integer***
  - `float`: ***floating point*** (also known as ***real***)
- Non-numeric
  - `char`: ***character***

```
#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.

But, 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).

C Built-In Boolean Data Type: `bool`

C also has a built-in data type for Booleans:

`bool`

The `bool` data type has possible values

`false`

and

`true`

However, some C compilers don't have the `bool` data type and the Boolean values `true` and `false` available by default; you have to make them available using this directive:

```
#include <stdbool.h>
```

(after `#include <stdio.h>`).

`bool` Data Type: Not Used in CS1313

In CS1313, we **WON'T** use the `bool` data type, nor its values `true` and `false`.

Instead, we'll use `char` or `int`.

Similarly, we'll use `0` for falsity and `1` (or any nonzero integer value) for truth.

Boolean Declaration

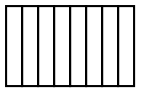
```
char CS1313_lectures_are_fascinating;
```

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.

Whether a `char` (or an `int`) is treated by a program as a Boolean or a character (respectively, an integer) **depends entirely on how you use it** in the program.

Boolean or Character Example #1

```
#include <stdio.h>

int main ()
{ /* main */
    const int maximum_short_height_in_cm = 170;
    const int program_success_code       =   0;
    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);
    return program_success_code;
} /* main */
```

Boolean or Character Example #2

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

Whether a `char` (or an `int`) is treated by a program as a Boolean or a character (respectively, an integer) **depends entirely on how you use it** in the program.

Boolean, Character or Integer? #1

In the previous example program, we had `char` variables named `I_am_Henry` and `I_am_tall`.

We treated them as Boolean variables in the calculation subsection, but in the output subsection we had:

```
printf("I_am_Henry = %d\n", I_am_Henry);
printf("I_am_tall = %d\n", I_am_tall);
```

How can this be?

Boolean, Character or Integer? #1

```
char I_am_Henry = 1;
char I_am_tall;
…
I_am_tall = (!I_am_Henry) || … ;
…
printf("I_am_Henry = %d\n", I_am_Henry);
…
printf("I_am_tall = %d\n", I_am_tall);
```

How can it be that the **same variable** is **simultaneously a Boolean, a character and an integer**?

It turns out that **`char`** not only means character, it also means an integer of 1 byte (8 bits).

This is confusing, but you'll get used to it.

Boolean Literal Constants

In C, a ***Boolean literal constant*** can have either of two possible values (but not both at the same time, of course):

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

Using Boolean Literal Constants #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 I_am_getting_a_bad_grade = 0;
```

- In an **assignment**:

```
this_is_my_first_guess = 1;
```

- In an **expression**:

```
Henry_is_short && 1;
```

Using Boolean Literal Constants #2

The first two of these uses – in a named constant declaration and in a variable declaration – are considered good programming practice, **AND SO IS THE THIRD** (in an assignment), which is a way that **Booleans are different from numeric data**.

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.

What is a Boolean Expression? #1

```
a || (b || c && !d) && e && (f || g) && h
```

In programming, a ***Boolean expression*** is a combination of:

- ***Boolean Operands***
- ***Boolean Operators***
- **Parentheses**: ( )

What is a Boolean Expression? #2

```
a || (b || c && !d) && e && (f || g) && h
```

In programming, a ***Boolean expression*** is a combination of:

- ***Boolean Operands***, such as:
  - Boolean literal constants (`0` for **false**, nonzero for **true**)
  - Boolean named constants
  - Boolean variables
  - ***Boolean-valued function invocations***
- ***Boolean Operators***
- **Parentheses**: ( )

What is a Boolean Expression? #3

```
a || (b || c && !d) && e && (f || g) && h
```

In programming, a ***Boolean expression*** is a combination of:

- ***Boolean Operands***
- ***Boolean Operators***, such as:
  - Relational Operators (which have **numeric operands**)
  - Logical Operators
- **Parentheses**: ( )

What is a Boolean Expression? #4

```
a || (b || c && !d) && e && (f || g) && h
```

In programming, a ***Boolean expression*** is a combination of:

- ***Boolean Operands***
- ***Boolean Operators***, such as:
  - Relational Operators (which have **numeric operands**)
    - Equal: ==
    - Not Equal: !=
    - Less Than: <
    - Less Than or Equal To: <=
    - Greater Than: >
    - Greater Than or Equal To: >=
  - Logical Operators
- **Parentheses**: ( )

What is a Boolean Expression? #5

```
a || (b || c && !d) && e && (f || g) && h
```

In programming, a ***Boolean expression*** is a combination of:

- ***Boolean Operands***
- ***Boolean Operators***, such as:
  - Relational Operators (which have **numeric operands**)
  - Logical Operators
    - ***Negation*** (NOT): !
    - ***Conjunction*** (AND): &&
    - ***Disjunction*** (OR): ||
- **Parentheses**: ( )

Boolean Expressions

Just like a numeric (arithmetic) expression, a ***Boolean expression*** is a combination of Boolean terms (such as variables, named constants, literal constants and Boolean-valued function calls), Boolean operators (for example, `!`, `&&`, `||`, 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	**Kind**	**Operator**	**Usage**	**Effect**
Identity	Unary	None	x	No change to value of x
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

C Boolean Expression Evaluation Values

C Boolean expressions evaluate to either:

- `0` (representing **false**)
- `1` (representing **true**)

Note that **any nonzero value represents true**, but, when C evaluates a Boolean expression, then if that expression evaluates to true, then specifically its value is `1`.

Note that **only `0` represents false, ever**.

Boolean Expression Example #1

```
#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 */
```

Boolean Expression Example #2

```
% gcc -o logic_expression_simple logic_expression_simple.c
% logic_expression_simple
 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
```

Boolean Variables Example #1

```
#include <stdio.h>

int main ()
{ /* main */
    const int true = 1;
    const int false = 0;
    int project_due_soon;
    int been_putting_project_off;
    int start_working_on_project_today;

    printf("Is it true that you have a programming project due
   soon?\n");
    printf("  (Answer %d for true, %d for false.)\n", true, false);
    scanf("%d", &project_due_soon);
    printf("Is it true that you have been putting off working on
   it?\n");
    printf("  (Answer %d for true, %d for false.)\n", true, false);
    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 */
```

Boolean Variables Example #2

```
% gcc -o pp_logic pp_logic.c
% pp_logic
Is it true that you have a programming project due soon?
  (Answer 1 for true, 0 for false.)
1
Is it true that you have been putting off working on it?
  (Answer 1 for true, 0 for false.)
1
Is it true that you should start working on it today?
ANSWER: 1
```

Relational Operations #1

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
```

Relational Operations #2

Operation	Operator	Usage	Result
Equal to	`==`	`x == y`	`1` if the value of `x` is exactly the same as the value of `y`; otherwise `0`
Not equal to	`!=`	`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`

Relational Expressions Example #1

```
#include <stdio.h>
int main ()
{ /* main */
    int CS1313_size, METR2011_size;
    printf("How many students are in CS1313?\n");
    scanf("%d", &CS1313_size);
    printf("How many students are in METR2011?\n");
    scanf("%d", &METR2011_size);
    printf("%d == %d: %d\n", CS1313_size, METR2011_size,
        CS1313_size == METR2011_size);
    printf("%d != %d: %d\n", CS1313_size, METR2011_size,
        CS1313_size != METR2011_size);
    printf("%d <  %d: %d\n", CS1313_size, METR2011_size,
        CS1313_size <  METR2011_size);
    printf("%d <= %d: %d\n", CS1313_size, METR2011_size,
        CS1313_size <= METR2011_size);
    printf("%d >  %d: %d\n", CS1313_size, METR2011_size,
        CS1313_size >  METR2011_size);
    printf("%d >= %d: %d\n", CS1313_size, METR2011_size,
        CS1313_size >= METR2011_size);
} /* main */
```

Relational Expressions Example #2

```
% gcc -o relational relational.c
% relational
How many students are in CS1313?
107
How many students are in METR2011?
96
107 == 96: 0
107 != 96: 1
107 <  96: 0
107 <= 96: 0
107 >  96: 1
107 >= 96: 1
```

Structure of Boolean Expressions

A Boolean expression can be long and complicated.

For example:

```
a || (b || c && !d) && e && (f || g) && h
```

Terms and operators can be mixed together in almost limitless variety, but they must follow these rules:

a unary operator has a term immediately to its right, and

a binary operator has terms on both its left and its right.

Boolean Expressions with Parentheses

Parentheses can be placed around any unary or binary subexpression:

```
(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."

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.

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

Boolean Precedence Order Example #1

```
!       0       ||      1

    1           ||      1
```

1

but

```
!       (0      ||      1)

!               1
```

0

Boolean Precedence Order Example #2

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

but

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

Boolean Precedence Order Example

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

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

Relational Expressions Example #1

```
#include <stdio.h>

int main ()
{ /* main */
    const int program_success_code = 0;
    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;
```

Relational Expressions Example #2

```
    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);
    return program_success_code;
} /* main */
```

Relational Expressions Example #3

```
% 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
```

Relational Expressions Example #4

```
% 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
```

Relational Expressions Example #5

```
% 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
```

Relational Expressions Example #6

```
% 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
```

Relational Expressions Example #7

```
% 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
```

Why Not Use `a < b < c`? #1

```
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;
```

Expressions like

`a < b < c` and `c < b < a`

WON'T accomplish what they look like they should.

Why not?

Why Not Use `a < b < c`? #2

Consider the expression `a < b < c`, and suppose that `a` is `6`, `b` is `5` and `c` is `4`; that is, `6 < 5 < 4`, which we know in real life is **false**.

But let's evaluate the expression as written.

1. Using the precedence rules, we evaluate left to right, so first we evaluate the subexpression `a < b`, which is a relational expression, so its result must be true (`1`) or false (`0`) – in this case false (`0`).
2. We then plug that result into the rest of the expression, getting `0 < c`; that is, `0 < 4`, which is **true** – so the value for `a < b < c` is wrong!

Instead, we need to use this: `(a < b) && (b < c)`

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, regardless of what the remaining terms evaluate to.

In that case, the program will stop bothering to evaluate the rest of the expression, because evaluating the rest of the expression wouldn't make any difference, but would **waste time**.

In such a case, we say that the Boolean expression will ***short circuit***: the rest of the expression won't be evaluated, because evaluating it would waste time, given that it won't change the result.

Short Circuit Example #1

```
#include <stdio.h>

int main ()
{ /* main */
    const int maximum_short_height_in_cm = 170;
    const int program_success_code       =   0;
    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);
    return program_success_code;
} /* main */
```

Short Circuit Example #2

```
% gcc -o short_circuit short_circuit.c
% short_circuit
I_am_Henry = 1
my_height_in_cm = 160
I_am_short = 1
my_middle_initial = J
```

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`.

Short Circuit Example #3

```
int  my_height_in_cm = 160;
char I_am_Henry = 1;
char I_am_short;
…
I_am_short =
    I_am_Henry ||
    (my_height_in_cm <
     maximum_short_height_in_cm);
…
```

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`.