Unit 9: Logical Expression

Learning Objectives

After this unit, students should:

- be able to read and write logical expressions in C using various logical operators, including `==`, `<`, `<=`, `<`, `>=`, `!=`, `&&`, `||` and `!`;
- be aware of the `bool` type and its values `true` and `false`, and the need to include `stdbool.h` to use it in C;
- be aware that in CS1010, we must never use `int` to indicate a true/false value;
- be aware of short-circuiting in logical expression;
- be aware of the CS1010 conventing of naming a boolean variable with the prefix `is_` or `has_`;
- be able to write a logical expression in appropriate order to exploit short-circuiting towards more efficient code

Representing a Boolean Value

You have seen some basic logical expressions in Unit 8. `n == 0`, `score >= 5`, `y > x`, are all logical expressions. They evaluate to either true or false.

We call a type that can contain either true or false as a Boolean data type, named after George Boole, a mathematician.

The Boolean data type in C is named `bool`. It can hold two values: `true` or `false`. All three of `bool`, `true`, and `false` are keywords introduced in modern C. To use them, you need to include the file `stdbool.h` at the top of your program.

Use `bool` is considered a cleaner way of representing true and false in C. Classically, C defines the numeric value 0 to be false, and everything else to be true. So, you can write code like this:

```
// x and y are long.
long is_diff = x - y;
if (is_diff) {
  cs1010_println_string("x and y store different values.");
}
```

The above is harder to understand and should be avoided. A cleaner way is to write:

```
bool is_diff = x != y;
if (is_diff) {
  cs1010_println_string("x and y store different values.");
}
```

Although not required by C, we will name a `bool` variable with a prefix `is_` or `has_` as a convention.

The code above can also be written as:

```
bool is_diff = x != y;
if (is_diff == true) {
  cs1010_println_string("x and y store different values.");
}
```

The comparison with `true` is redundant, however, and should be skipped.

Logical Operators

Just like we can perform arithmetic operations on integers and real numbers, we can perform logical operations on boolean values. These allow us to write complex logical expressions.

Consider the example problem: Write a function that, given the birth year of a person, determine if he or she belongs to Generation Z, defined as someone whose birth is between 1995 and 2005, inclusive.

We can write the function as follows using what we have known:

```
bool is_gen_z(long birth_year)
{
  if (birth_year >= 1995) {
    if (birth_year <= 2005) {
      return true;
    }
  }
  return false;
}
```

To be in Generation Z, both conditions `birth_year >= 1995` and `birth_year <= 2005` must be true. We can use the logical AND `&&` operator to simplify the code above to:

```
bool is_gen_z(long birth_year)
{
  if ((birth_year >= 1995) && (birth_year <= 2005)) {
```

```
    return true;
  }
  return false;
}
```

or simply:

```
bool is_gen_z(long birth_year)
{
  return ((birth_year >= 1995) && (birth_year <= 2005));
}
```

The AND operator, `&&`, evaluates to true if and only if both operands are true.

> **Common Error**
>
> A common mistake by a new C programmer is to write `1995 <= birth_year <= 2005` as the logical expression. Unfortunately, in C, we cannot chain the comparison operators together.

What if we want to write a function to determine if someone is NOT part of Generation Z? This means that they are born *either* before 1995 *or* after 2005. To have an expression that evaluates to true if either one of two expressions is true, we can use the OR operator, `||`.

```
bool is_not_gen_z(long birth_year)
{
  return ((birth_year < 1995) || (birth_year > 2005));
}
```

Generally, we prefer to write functions that check for the positives, as it is generally easier to think in terms of the positives. So the example `is_not_gen_z` above is for illustration purposes only, we do not encourage you to write functions that check for the negatives. In any case, if we want to check if someone is not a Generation Z, we can use the `!` NOT operator.

```
if (!is_gen_z(birth_year)) {
    :
}
```

This operator can be used as part of the boolean expression:

```
bool is_not_gen_z(long birth_year)
{
  return !((birth_year >= 1995) && (birth_year <= 2005));
}
```

Short-Circuiting

When evaluating the logical expressions that involve `&&` and `||`, C uses "short-circuiting". This means that, if we already know, for sure, that a logical expression is true or is false, there is no need to continue the evaluation. The corresponding `true` and `false` value will be returned.

Consider the following:

```
bool is_gen_z(long birth_year)
{
  return ((birth_year >= 1995) && (birth_year <= 2005));
}
```

If the argument `birth_year` is `1970`, then, the expression `(birth_year >= 1995)` already evaluates to `false`, and the whole statement is false. We do not need to evaluate the second expression `(birth_year <= 2005)`.

Similarly, for

```
bool is_not_gen_z(long birth_year)
{
  return ((birth_year < 1995) || (birth_year > 2005));
}
```

When `birth_year` is `1970`, the expression `(birth_year < 1995)` is `true`, so we know that the whole statement is `true`. There is no need to check if `(birth_year > 2005)`.

In both examples above, the savings due to short-circuiting is not much -- since we are comparing two numbers, and there is no *side effects* in comparing `birth_year` to `2005`. But, let's suppose that we introduce two functions with side effects (of printing to the screen):

```
bool not_too_old(long birth_year)
{
  if (birth_year >= 1995) {
    cs1010_print_string("not too old..");
    return true;
  }
  cs1010_print_string("too old..");
  return false;
}

bool not_too_young(long birth_year)
{
  if (birth_year <= 2005) {
    cs1010_print_string("not too young..");
    return true;
```

```
16    }
17    cs1010_print_string("too young..");
18    return false;
19  }
20
21  bool is_gen_z(long birth_year)
22  {
23    return not_too_old(birth_year) && not_too_young(birth_year);
24  }
```

When we call `is_gen_z(1984)`, you might expect `too old..not too young..` to be printed, but due to short-circuiting, the code only prints `too old..`.

Another reason to keep short-circuiting in mind is that the order of the logical expressions matter: we would want to put the logical expression that involves more work in the second half of the expression. Take the following example:

```
1  if (number < 100000 && is_prime(number)) {
2      :
3  }
```

for efficiency

- plan ahead and compare the things want to check

Checking whether a number is below 100,000 is easier than checking if a number is prime. So, we can skip checking for primality if the `number` is too big. Compare this to:

```
1  if (is_prime(number) && number < 100000) {
2      :
3  }
```

→ expensive function call

Suppose `number` is a gigantic integer, then we would have spent lots of effort checking if `number` is a prime, only to find out that it is too big anyway!

Problem Sets

Problem 9.1

Given two `bool` variables, `a` and `b`, there are four possible combinations of `true` `false` values. What are the values of `a && b`, `a || b`, and `!a` for each of these combinations? Fill in the table below.

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

Problem 9.2

Consider the function below, which aims to return the maximum value given three numbers.

```
long max_of_three(long a, long b, long c)
{
  long max = 0;
  if ((a > b) && (a > c)) {
    // a is larger than b and c
    max = a;
  }
  if ((b > a) && (b > c)) {
    // b is larger than a and c
    max = b;
  }
  if ((c > a) && (c > b)) {
    // c is larger than a and b
    max = c;
  }
  return max;
}
```

(a) What is wrong with the code above?

(b) Give a sample test value of `a`, `b`, and `c` that would expose the bug.

(c) Fix the code above to remove the bug.

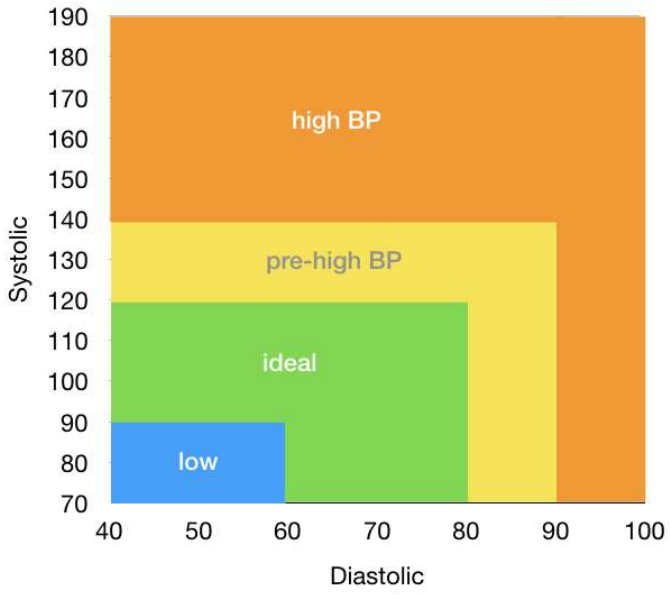
(d) Replace the three `if` statements in the code above with `if-else` statements. Draw the corresponding flowchart.

Problem 9.3

Write a function that takes in a blood pressure measurement, and prints either `low`, `ideal`, `pre-high`, and `high` depending on the input values. The blood pressure is given as

two `long` values, the systolic and the diastolic. The text to be printed depends on the range, depicted in the figure below.

```
void print_blood_pressure(long systolic, long diastolic)
{
  :
}
```



The figure does not say how to classify the data if the values fall exactly on the boundary of two regions. In this case, you can classify it into either region.