Unit 8: Conditional Statement

Learning Objectives

After going through this unit, students should:

- understand that branching in flowcharts relates to either `if-else` statements or loop;
- be able to identify the three components of an `if-else` statement: the condition, the `true` block, and the `false` block;
- be able to develop an `if-else` statement in C, including, (i) nested `if-else`, (ii) `if` without `else` blocks , and (iii) `else if` blocks;
- be able to model a computational solution with different conditions as a table;
- understand the meaning and the use of the type `void`
- understand the term *string* and be able to use `cs1010_println_string` to print a string to the screen.
- be aware that in CS1010:
- curly brackets must not be skipped even if a block contains only a single statement;
  - conditional operators are banned.
- be aware of floating-point numbers must not be compared using equality.

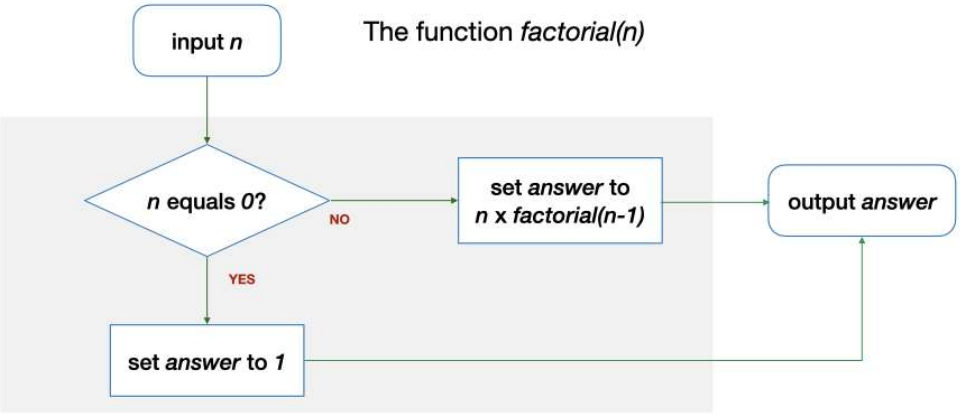
Branching

So far the C programs that we have written have a straightforward *execution path*. The execution flows from top to bottom in `main`, jumping to a function being called (or *callee*), and back to the caller when the function returns.

We have, however, seen a few examples so far where the execution path can *branch* off to either one of two paths, depends on a condition:

- In the algorithm to compute the $max(L, k)$, we check if i equals k, and stop checking if it is true; repeat if it is false.
- In the algorithm to compute the $max(L, k)$, we check if $l_i > m$, and update m only if this is true.
- In the algorithm to compute the $factorial(n)$, we check if n equals 0, and return 1 if it is true, otherwise, we return $n \times factorial(n-1)$.

We are not ready to write C code that processes a list repetitively yet. So the first example above will be covered in Unit 11. In this unit, we will use the $factorial(n)$ function as an example.



In C, the $factorial(n)$ would look like this:

```c
long factorial(long n)
{
  long answer;
  if (n == 0) {
    answer = 1;
  } else {
    answer = n * factorial(n - 1);
  }
  return answer;
}
```

In this example, you see two new C keywords `if` and `else`. These keywords are used *conditional blocks*. The general syntax is:

```
if (<logical expression>) {
  "true block": statements to be executed if expression evaluates to true
} else {
  "false block": statements to be executed if expression evaluates to true
}
```

The `if` keyword is followed by a *logical expression* in parenthesis. This is followed by a block of statements (in curly braces `{` and `}`). If the logical expression is true, then the statements are executed, otherwise, they are skipped, and the statements following the `else` block is executed instead. For this reason, the group of statements following `if` is known as a *true block*, and the group of statements following `else` is known as the *false block*.

Comparison Operator

The logical expression `n == 0` is true if the variable `n` holds the value of `0`.

> **== vs =**
>
> Note that the use of TWO `=` signs. This is often confused by newbie programmers with a single `=` sign, which is used for assignment. A common bug is to write
>
> ```
> 1  if (n = 0) {
> 2     :
> 3  }
> ```

The `==` is known as a *comparison operator*. It compares if the left-hand side has the same value as the right-hand side. Other comparison operators include `>` (greater than), `<` (less than), `>=` (greater or equal to), `<=` (less than or equal to), and `!=` (not equal).

In other words, the variable `answer` will be set to `1` if the parameter `n` equals `0`. What if `n` is not `0`? The block that contains Line 5 `return 1;` will be skipped, and Line 7 `answer = n * factorial(n - 1);` will be executed instead.

Nested Else-If

The example above considers *two* possible execution paths only. In some situations, we may need to consider more than two execution paths. Take the following problem for example. You are given the numerical score for an assignment, ranged between 0 and 10. Print out the letter grade of the assignment according to the table below:

Score	Letter Grade
8 or higher	A
Less than 8 but 5 or higher	B
Less than 5 but 3 or higher	C
Less than 3	D

Since the `if-else` statement only allows branching into two possibilities, we can branch into multiple possibilities by nesting the `if-else` statements hierarchically. We can first break the table down into three tables, each containing only two rows, with one row a negation of the other row.

Score	Letter Grade
8 or higher	A
Less than 8	See Table 1

Table 1 (less than 8)

Score	Letter Grade
5 or higher	B
Less than 5	See Table 2

Table 2 (less than 5)

Score	Letter Grade
3 or higher	C
Less than 3	D

The tables above can then be written into the following function:

```
1  void print_score(double score)
2  {
3    if (score >= 8) {
4      cs1010_println_string("A");
5    } else {
6      // Table 1
7      if (score >= 5) {
8        cs1010_println_string("B");
9      } else {
10       // Table 2
11       if (score >= 3) {
12         cs1010_println_string("C");
13       } else {
14         cs1010_println_string("D");
15       }
16     }
17   }
18 }
```

There are three nested `if-else` in the function above. Note how I use indentation to clearly indicate the nesting of blocks. Such nesting or indentation is not required by C standard, but is a commonly accepted coding practice, and is required for CS1010.

The code below compiles perfectly but is not as easy to read by a human as the above.

```
void print_score(double score)
{
  if (score >= 8) {
    cs1010_println_string("A"); } else { // Table 1
    if (score >= 5) { cs1010_println_string("B");
  } else { // Table 2
if (score >= 3) {
  cs1010_println_string("C");
      } else {
        cs1010_println_string("D");
    } } } }
// Don't write code like this.
```

There are also a couple of "first" in the sample code above:

- You see the keyword `void` for the first time. `void` is a special type that indicates nothing. The function `print_score` does not return anything, it accepts an input `score` and print something to screen. As such, we say that the return type of `print_score` is `void`.
- You see *strings* for the first time (`"A"`, etc.). A *string* basically is a sequence of characters. We use double quotes `"` to mark the beginning and the end of a string, and use the CS1010 I/O library function `cs1010_println_string` to print a string to the screen.

You can imagine that as the number of possible letter grades increases (NUS has 11), we will have many nested `if-else`, and the code gets complicated. To reduce the number of nesting, we can write `else if` directly, without nesting:

```
void print_score(double score)
{
  if (score >= 8) {
    cs1010_println_string("A");
  } else if (score >= 5) {
    cs1010_println_string("B");
  } else if (score >= 3) {
    cs1010_println_string("C");
  } else {
    cs1010_println_string("D");
  }
}
```

The above code is easier to read but has exactly the same flow as the one with nested `if-else` earlier.

Avoid Skipping the Curly Braces

The C standard says that, if the block contains only one statement, we can skip the curly braces `{` and `}`. In the example above, we can write:

```
void print_score(double score)
{
  if (score >= 8)
    cs1010_println_string("A");
  else if (score >= 5)
    cs1010_println_string("B");
  else if (score >= 3)
    cs1010_println_string("C");
  else
    cs1010_println_string("D");
}
```

Despite being allowed by the C standard, this is considered a bad practice and should be avoided. Imagine some time later, you go back to this code, and want to write something extra:

```
void print_score(double score)
{
  if (score >= 8)
    cs1010_println_string("A");
  else if (score >= 5)
    cs1010_println_string("B");
  else if (score >= 3)
    cs1010_println_string("C");
  else
    cs1010_println_string("You can do better!");
    cs1010_println_string("D");
}
```

What would be printed?

The famous Apple `goto fail` bug wouldn't have happened in the there is a pair of curly braces added!

Alternatively, if you have code like this:

```
if (score >= 8)
  if (late_penalty != 0)
      cs1010_println_string("late submission");
else
  cs1010_println_string("you can do better!");
```

It might look like `you can do better!` will be printed if `score` is less than 8, but actually, `you can do better!` will be printed if the `score` is larger or equal to 8 and there is no late penalty, which is not what is intended.

Skipping `else`

In some algorithms, it is not necessary for us to have an else block.

Take, for instance, suppose we have a variable `max_so_far` and we wish to compare it with another variable `x`. We will set `max_so_far` to `x` if `x` is the larger of the two, but we do not need to do anything otherwise.

We could write it as:

```
  :
if (max_so_far < x) {
  max_so_far = x;
} else {
}
  :
```

The false block is empty, and we could keep our code succinct by removing it altogether.

```
  :
if (max_so_far < x) {
  max_so_far = x;
}
  :
```

One, however, needs to be careful with skipping the else block.

Let's consider another example. Suppose we have three `long` variables, `x`, `y`, and `max`, and we want to set `max` to the maximum of `x` and `y`.

Consider the following code snippet:

```
if (x > y) {
  max = x;
}
if (x < y) {
  max = y;
}
```

Take a moment to understand the code above, and see if you can figure out what is wrong.

plan out all scenarios

When we think about writing conditionals, we have to exhaustively reason about what are all the possible scenarios that could occur. In this example, we need to think about what are the possible relationships between `x` and `y` when we compare `x` an `y`. There are actually three possibilities!

- `x > y`: in this case, `x` is larger and we set `max` to `x`

- `y > x`: in this case, `y` is larger and we set `max` to `y`
- `x == y`: in this case, both are equally large, so the maximum of the two can be either `x` or `y`.

In the code above, `max` is not set properly if `x == y`!

The following code adds the third case and arbitrarily chooses to set `max` to `y` if both `x` and `y` have the same value.

```
if (x > y) {
  max = x;
}
if (x < y) {
  max = y;
}
if (x == y) {
  max = y;
}
```

The code snippet above now correctly sets `max` to the maximum of `x` and `y`. The code, however, is not very satisfying, since we compare between `x` and `y` three times! Since the "true block" for `x < y` and `x == y` are the same, and we can combine it into a single comparison `x <= y`.

```
if (x > y) {
  max = x;
}
if (x <= y) {
  max = y;
}
```

But, if `x > y` is false, then `x <= y` must be true! We say that `x > y` and `x <= y` are *negation* (or opposite) of each other. So, the check for `x <= y` is redundant -- checking `x > y` is enough to tell us if `x <= y`. We can re-write the code above succinctly as:

```
if (x > y) {
  max = x;
} else {
  max = y;
}
```

So, we should not skip the `else` block in the case above to begin with.

There are, however, other cases where the `else` block can be skipped. Consider the code:

```
long factorial(long n)
{
  long answer;
```

```
4    if (n == 0) {
5      answer = 1;
6    } else {
7      answer = n * factorial(n - 1);
8    }
9    return answer;
10 }
```

We could actually write it more succinctly as:

```
1  long factorial(long n)
2  {
3    if (n == 0) {
4      return 1;
5    } else {
6      return n * factorial(n - 1);
7    }
8  }
```

We can have a more succinct code by removing the variable `answer` since it is not used in any other meaningful way except as a placeholder to be returned later.

Now that we removed the variable `answer`, we could go one step further, and remove the `else`.

```
1  long factorial(long n)
2  {
3    if (n == 0) {
4      return 1;
5    }
6    return n * factorial(n - 1);
7  }
```

We can always do this without changing the outcome of the function since Line 6 will be executed only if the condition of Line 3 is false. If the condition on LIne 3 is true, the function will return and Line 6 will be skipped!

Conditional Operator

The conditional operator consists of two special characters `?` and `:` and is used in the format of:

```
1  condition ? true expression : false expression;
```

If the `condition` evaluates to true, then the `true expression` will be evaluated and returned, otherwise, the `false expression` will be evaluated and returned.

The conditional operator allows us to replace

```
if (x > y) {
  max = x;
} else {
  max = y;
}
```

with a single line:

```
max = (x > y) ? x : y;
```

We can nest the conditional operator as well, but it does not necessarily make your code easier to read once you start nesting them up. We do not encourage you to nest the conditional operator in CS1010 and to limit its usage to simple cases above.

Comparing Real Numbers

Recall that we said real numbers cannot be represented exactly in computers. Comparing real numbers, therefore, becomes a little bit trickier in programming. The `if` statement

```
double expected_value = 0.3;
double sum = 0.1 + 0.2;
if (sum == expected_value) {
 :
}
```

would not be evaluated as `true` as expected!

Thus, to compare real numbers, we normally allow some errors in comparisons -- we want the absolute difference between `sum` and `expected_value` to be small enough.

```
double expected_value = 0.3;
double sum = 0.1 + 0.2;
if (fabs(sum - expected_value) < 0.000001) {
 :
}
```

depends on the required accuracy

fabs is to find the absolute |2-3| = 1

Problem Sets

Problem 8.1

Do the following two functions behave the same way? Explain.

```
```

```
long factorial(long n)
{
  long answer;
  if (n == 0) {
    answer = 1;
  }
  answer = n * factorial(n - 1);
  return answer;
}
```

and

```
long factorial(long n)
{
  long answer;
  if (n == 0) {
    answer = 1;
  } else {
    answer = n * factorial(n - 1);
  }
  return answer;
}
```

Problem 8.2

Draw the flowcharts for the three code snippets below.

(a)

```
if (x > y) {
  max = x;
}
if (x < y) {
  max = y;
}
if (x == y) {
  max = y;
}
```

(b)

```
if (x > y) {
  max = x;
} else if (x < y) {
  max = y;
} else if (x == y) {
  max = y;
}
```

(c)

```
if (x > y) {
  max = x;
} else {
  max = y;
}
```

Problem 8.3

Suppose we break down the table below in a slightly different way.

Score	Letter Grade
8 or higher	A
Less than 8 but 5 or higher	B
Less than 5 but 3 or higher	C
Less than 3	D

We rewrite the tables into three smaller tables, as:

Score	Letter Grade
5 or higher	See Table 3
Less than 5	See Table 4

where Table 3 (5 or higher) is

Score	Letter Grade
8 or higher	A
Less than 8	B

and Table 4 (less than 5) is

Score	Letter Grade
3 or higher	C
Less than 3	D

Write the corresponding `if-else` statements to print out the letter grade based on the tables above.