03-Conditional Statements

In the last section, we learned how to use read and write data from and to the system, as well as how to declare variables which store program information such as user input. In this section, we will learn how to write programs that evaluate user input and how we can write programs whose output and behavior depends on it.

Control Flow

In computer science, **control flow** refers to the way in which programs are executed by a machine. The control flow of a program is determined via ***conditional statements***, which are used to determine what segments of your program should be executed based on **logical conditions**.

Before we get into how control flow affects a program, it's important that we understand what ***booleans*** are.

Booleans

A **boolean** is a type of variable (denoted in C++ with `bool`) which represents a logical value that is either `true` or `false`. We can define boolean variables as such:

```
bool ketchup_on_steak = true;
bool pineapple_on_pizza = false;
```

On their own, boolean variables are not very useful, particularly if they are not set based on user/data input. We can assign the value of a boolean to the result of a ***logical expression***. A logical expression returns either `true` or `false` based on the logic of said statement. For example:

```
int x = 5;
int y = 2;

int this_is_false = x == y;
int this_is_true = x > y;
int this_is_false = x < y;
int this_is_true = x - y == 3;
```

In the above example, the operators `==`, `>`, and `<` are used to compare numerical values. These are called **logical operators** and are used to evaluate boolean expression.

Operator	Syntax	Usage
`==`	`x==y`	Compares two values and returns `true` **if and only if** `x` and `y` are equal. **Do NOT confuse with `=`, which is the assignment operator.**
`!=`	`x!=y`	Logical inversion of `==`. Compares two values and returns `false` **if and only if** `x` and `y` are **NOT** equal.
`>`	`x>y`	Compares two values and returns `true` **if and only if** `x` is **greater than** `y`.
`<`	`x<y`	Returns `true` **if and only if** `x` **is less than** `y`.
`>=`	`x>=y`	Returns `true` if `x > y` OR `x == y`.
`<=`	`x<=y`	Returns `true` if `x < y` OR `x == y`

> The operators `<`, `>`, `>=` and `<=` are called **relational operators**. `==` and `!=` are called **equality operators**.

Compound Logical Operators

Like many other statements, logical statements can be **compound** to form complex logical expressions. Logical expressions are conjoined via AND, OR, and NOT expressions:

- `||` **operator** - `x || y` returns `true` if `x` is `true` OR `y` is `true`.
- `&&` **operator** - `x && y` returns `true` **if and only if** `x` is true AND `y` is true.
- `!` **operator** - `!x` returns the **logical inversion** of `x`.
  - if `x == true`, `!x == false`.
  - if `x == false`, `!x == true`.

Operator Precedence (Order of Operations)

It is important to understand the order of operations of complex logical expressions so that the meaning of an expression is intuitive/readable.

In C++, logical expressions are evaluated left-to-right. In the expression:

```
x == y && x == z;
```

The sub-expression `x == y` is evaluated before `x == z`. This is also the case for `||`, and the equality and relational operators.

Simple two-operand arithmetic expressions take precedence over logical operators, so in the expression:

```
x + y > z && y == z / y;
```

`x + y` is evaluated before `> z`, and may be expressed as `(x + y) > z`. Same with `z / y`.

You may use parentheses within expressions to make the precedence of your expression clear:

```
(x + y) > z && y == (z / y);
```

> Using parentheses to make operator precedence clear in your code is good practice, as it improves readability and prevents bugs caused by slight oversights in understanding and implementing operator precedence.

For advanced readers:

> You can view the entire list of operators and their precedence at https://en.cppreference.com/w/cpp/language/operator_precedence.

`if/else` Statements

An ***`if`*** statement is a statement that executes a *block* of code if its **conditional statement** is **`true`**. The structure of an `if` statement is as follows:

```
if (conditional statement) {
        // conditional code
}
```

> Note that conditional statements inside the parentheses of an `if` block do not need to end with a semicolon.

Conversely, we can execute a sequence of code when a condition is ***not*** `true`:

```
if (conditional statement) {
        // runs if <conditional statement> is true
} else {
        // runs if <conditional statement> is false
}
```

You may also chain `if/else` statements to perform various blocks of code based on a series of conditions:

```
if (conditional statement) {
        // runs if <conditional statement> is true
} else if (other conditional statement) {
```

```
        // runs if <conditional statement> is false and
        // <other conditional statement> is true
} else {
        // runs if both <conditional statement> and
        // <other conditional statement> are false.
}
```

For advanced readers:

> `if () {} else if () {}` is semantically equivalent to:
>
> ```
> if (...) {
>     ...
> } else {
>     if (...) { ... }
> }
> ```
>
> This is because `if` statements followed by *a single statement of execution* does not need to be surrounded by braces. So:
>
> ```
> if (x == y) std::cout << "x is equal to y.\n";
> else std::cout << "x is not equal to y.\n"
> ```
>
> is a valid conditional expression.

As an example, let's write a program which takes a user's age as input, and determines whether or not they are of legal age to own a driver's license in the United States:

```
#include <iostream>

int main() {
        int age;
        std::cout << "Please enter your age: ";
        std::cin >> age;

        if (age >= 18) {
                std::cout << "You are able to own a driver's license.\n";
        } else if (age >= 16) {
                std::cout << "You are likely able to own a provisional
license.\n";
        } else {
                std::cout << "You are likely unable to drive, legally.\n";
        } // Don't take my word for this, consult your state's laws
}
```

Scope

You might notice the usage of *curly braces/brackets* around the `//conditional code` segment. **Code that is surrounded by curly braces is said to be in a certain *scope***. For example, in our "hello world" program:

```
#include <iostream>

int main() {
        std::cout << "Hello, world!\n";
        return 0;
}
```

The segment

```
        std::cout << "Hello, world!\n";
        return 0;
```

is said to be in the scope of `main()`.

Exercise

As a comprehensive exercise, write a program which **asks the user to enter** their **name** and **a percentile grade (weighted 0-100%)** and displays the **letter grade** of the student's assignment to the console:

Percentile Range	Letter Grade
`90-100%`	**A**
`80-89%`	**B**
`70-79%`	**C**
`60-69%`	**D**
`0-59%`	**F**

The output should look something like:

```
Please enter your name: Sarah Evans
Please enter your assignment grade: 86
Sarah Evans, you've received a B on your assignment.
```

What's Next?

In the next section, we will learn how to use conditional statements to repeat segments of code using *loops*.

Courtesy of *DeCentralize the Web* (2024)