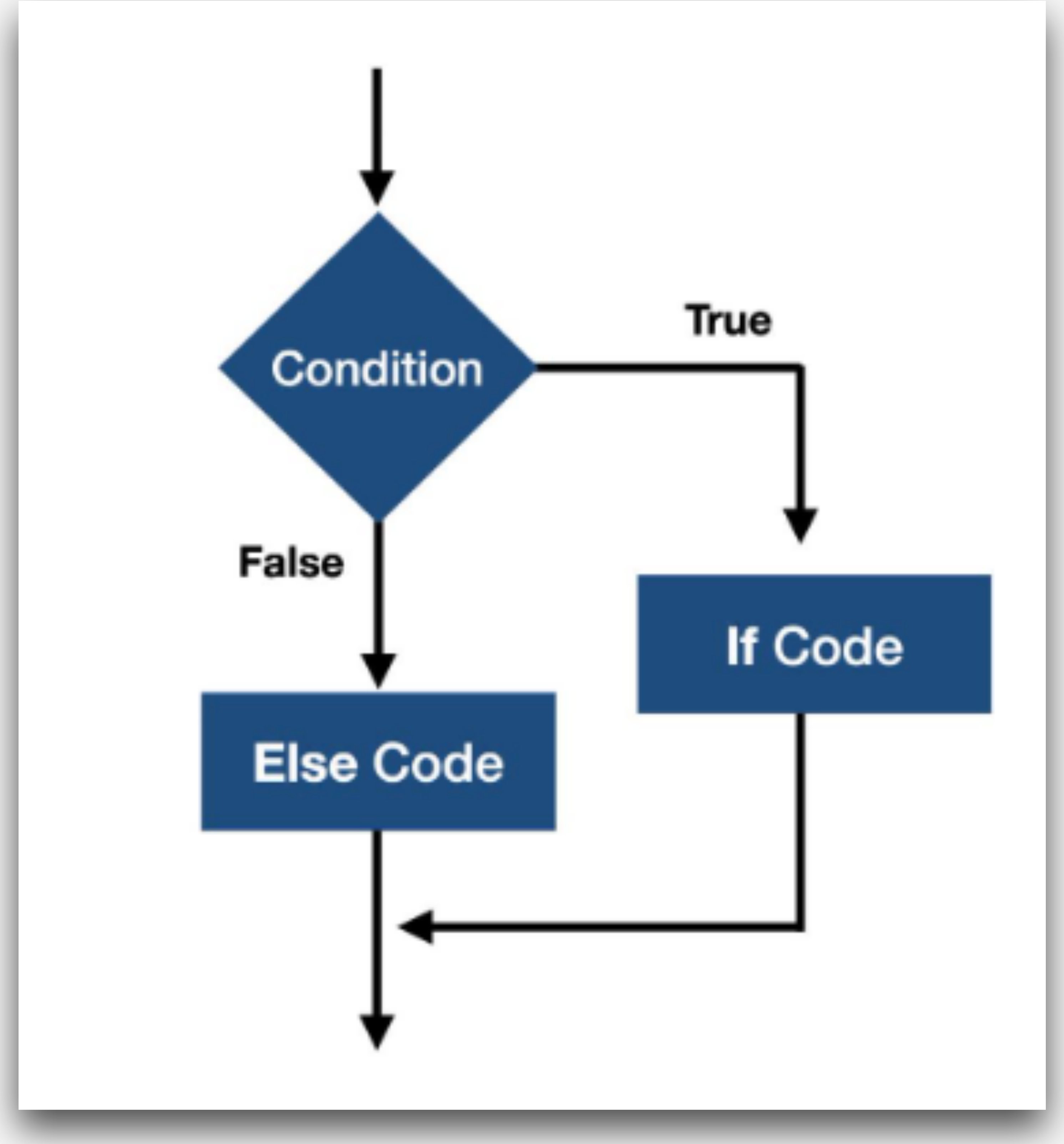
Chapter 3 & 4

Branches and Loops

Branching Basics

If, if-else, and switch statements



If Statement

If statements execute a block of code if and only if the expression evaluates to ***true.***

Syntax:

```
if (/*condition*/) {
   // Block of code
}
```

Example:

```
if (grade >= 70) {
  cout << “Pass"
}
```

Easy! This is just like Java

If-Else

Use the ***else*** statement to indicate a block of code that is executed if the ***if*** statement evaluates to ***false.***

Syntax:

```
if (/*condition*/) {
   // Block of code
} else {

}
```

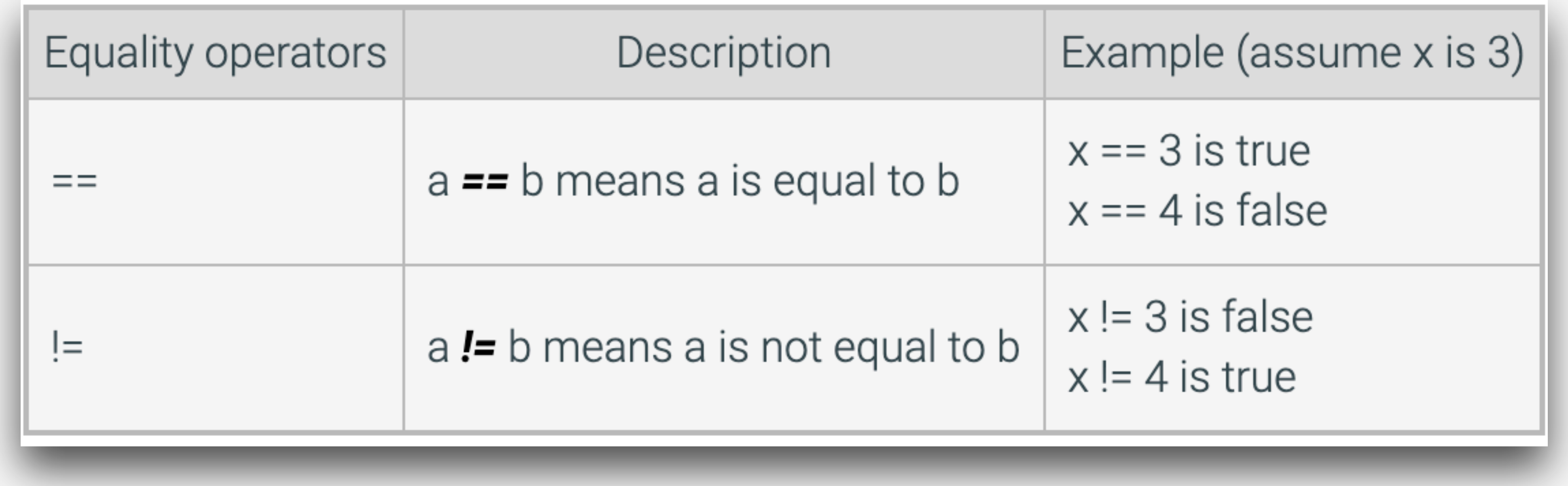
Example:

```
if (grade >= 70) {
  cout << "Pass"
} else {
  cout << "Fail"
}
```

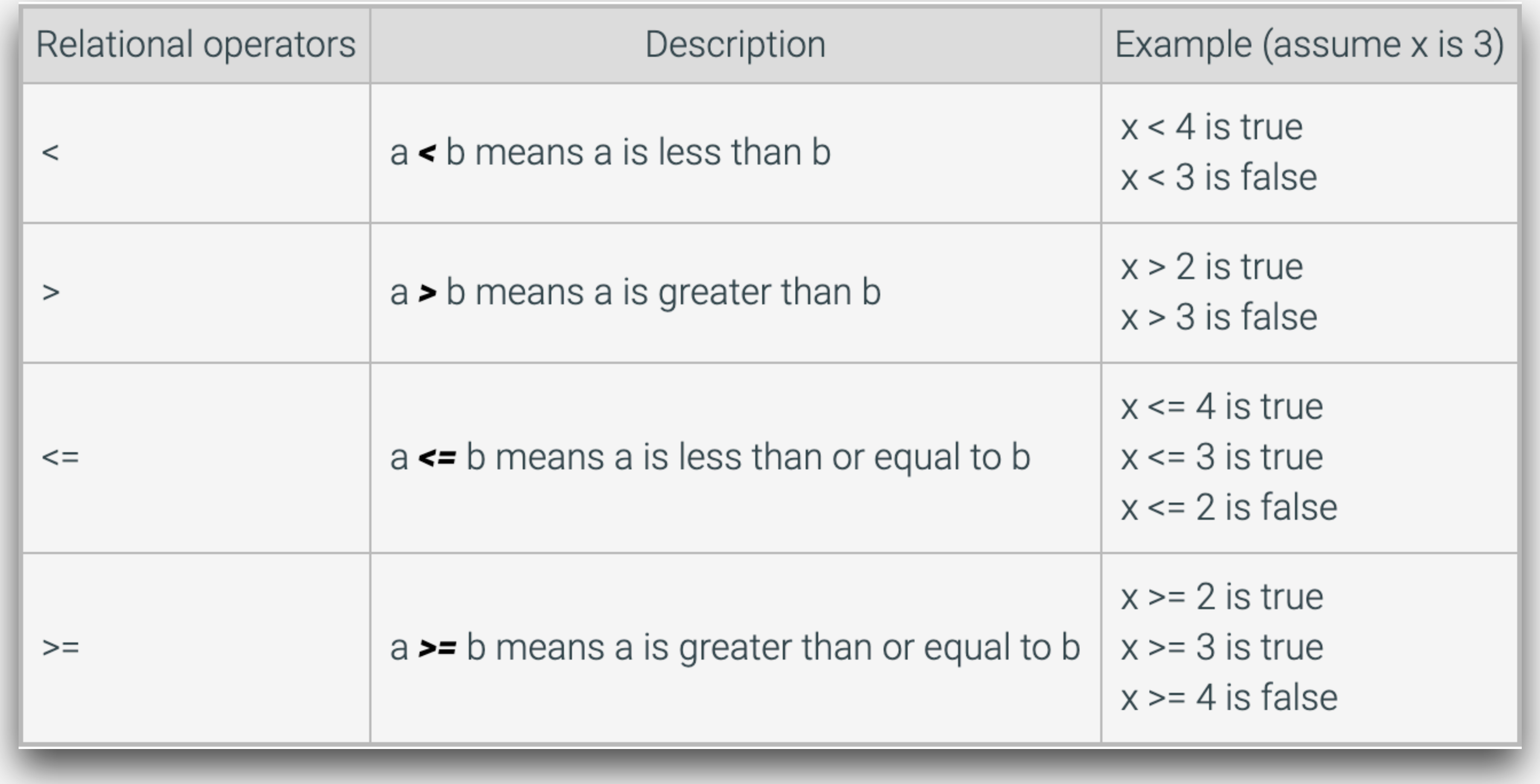
Relational Operators

Relational Operators are binary operators that are used to compare how one operands value compares to another.

- We frequently use ***relational*** operators in ***if-else*** statements, however they are not required.



Equality operators	Description	Example (assume x is 3)
==	a **==** b means a is equal to b	x == 3 is true x == 4 is false
!=	a ***!=*** b means a is not equal to b	x != 3 is false x != 4 is true



Relational operators	Description	Example (assume x is 3)
<	a **<** b means a is less than b	x < 4 is true x < 3 is false
>	a **>** b means a is greater than b	x > 2 is true x > 3 is false
<=	a **<=** b means a is less than or equal to b	x <= 4 is true x <= 3 is true x <= 2 is false
>=	a **>=** b means a is greater than or equal to b	x >= 2 is true x >= 3 is true x >= 4 is false

examples: golf.cpp

Logical Operators and Conditionals

Logical Operators are used with expressions to yield a boolean result.

`!`	logical NOT.	Ex: `!(1 == 1)`
`&&`	logical AND.	Ex: `(true && true)`
`||`	logical OR.	Ex: `(true || false)`

- If the first AND fails, the second expression is not evaluated.

```
int grade = 89;
if (grade > 90 && grade < 93)
```

- If the first OR passes, the second expression is not evaluated.

```
int day = 6;
if (day == 6 || day == 7)
```

This is called *short circuiting*

Ternary Operator

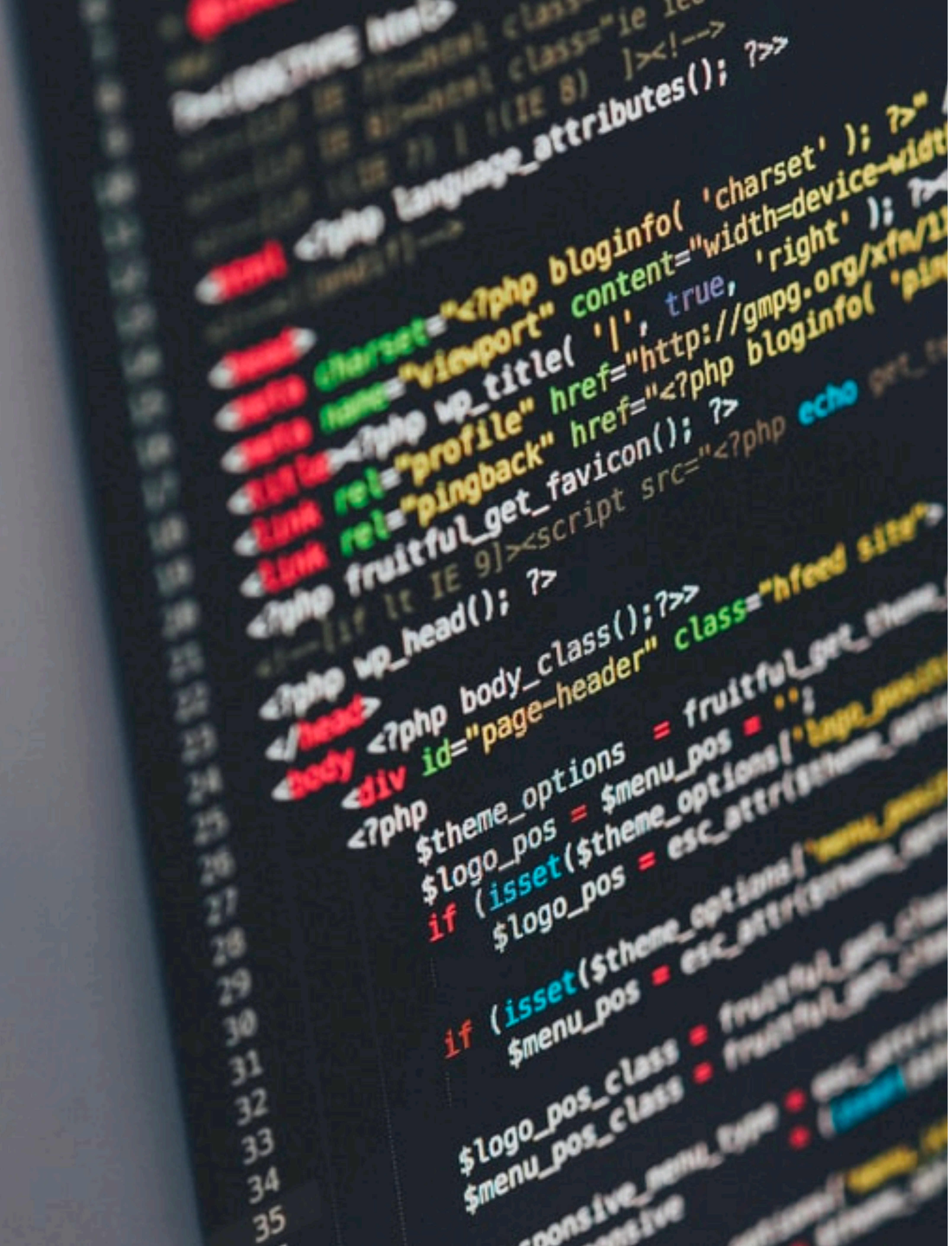
A ***ternary operator*** evaluates the test condition and executes a block of code based on the result of the condition

- Ternary Operator can be used in place of ***if-else*** in certain scenarios

```
(conditional) ? <value_if_true> : <value_if_false>;
```

- Although possible, DO NOT use nested ternary operators, as this makes your code hard to read!

examples: ternary.cpp



Challenge!

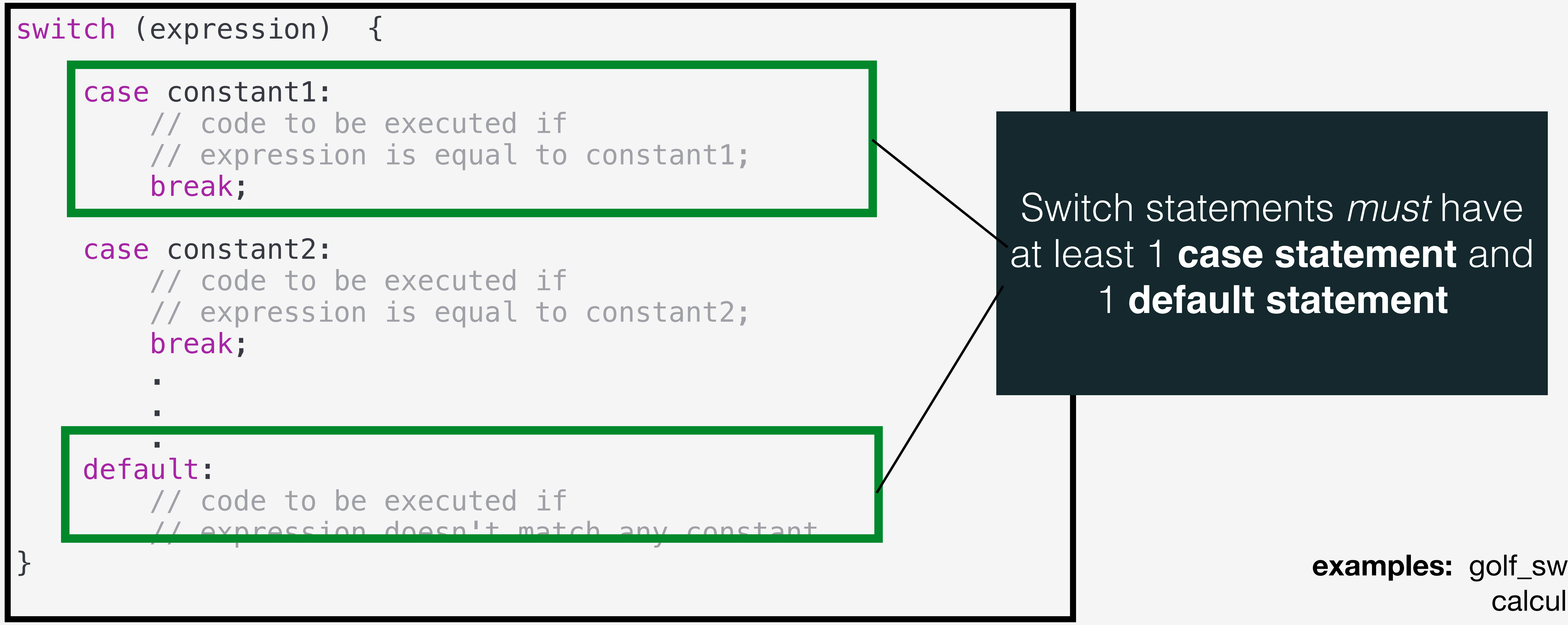
ZyBook Lab 3.24

https://learn.zybooks.com/zybook/SMUCS1342Spring2023/chapter/3/section/24

Switch Statement

A switch statement is an alternative to if-else statements that allows programs to execute a block of code among many alternatives by comparing a value to an integer expression

- The ***case*** must be a constant integral or integral expression (Determined at compile time)



```
switch (expression)  {

    case constant1:
        // code to be executed if
        // expression is equal to constant1;
        break;

    case constant2:
        // code to be executed if
        // expression is equal to constant2;
        break;
        .
        .
        .
    default:
        // code to be executed if
        // expression doesn't match any constant
}
```

Switch statements *must* have at least 1 **case statement** and 1 **default statement**

examples: golf_switch.cpp
calculator.cpp

Loops

for, while, do-while

While Loops

A construct that repeatedly executes a block of sub-statements while the loop condition evaluates to true

- While loops are ***pre-test*** loops
- Each loop of execution is called an ***iteration***
- If the condition is ***false*** midway through an iteration, the while loop will complete the iteration before terminating

```
int count = 0;
while (count < 10) {
   cout << count << endl;
   count++;
}
```

While Loops w/ user input

While loops can be a good way to repeatedly run the same code for users to simulate a state machine or a menu driven program.

Example: Write a program that calculates the sum of each number given by a user, then prints out the average of those numbers to the console whenever a user enters the value ***0***.

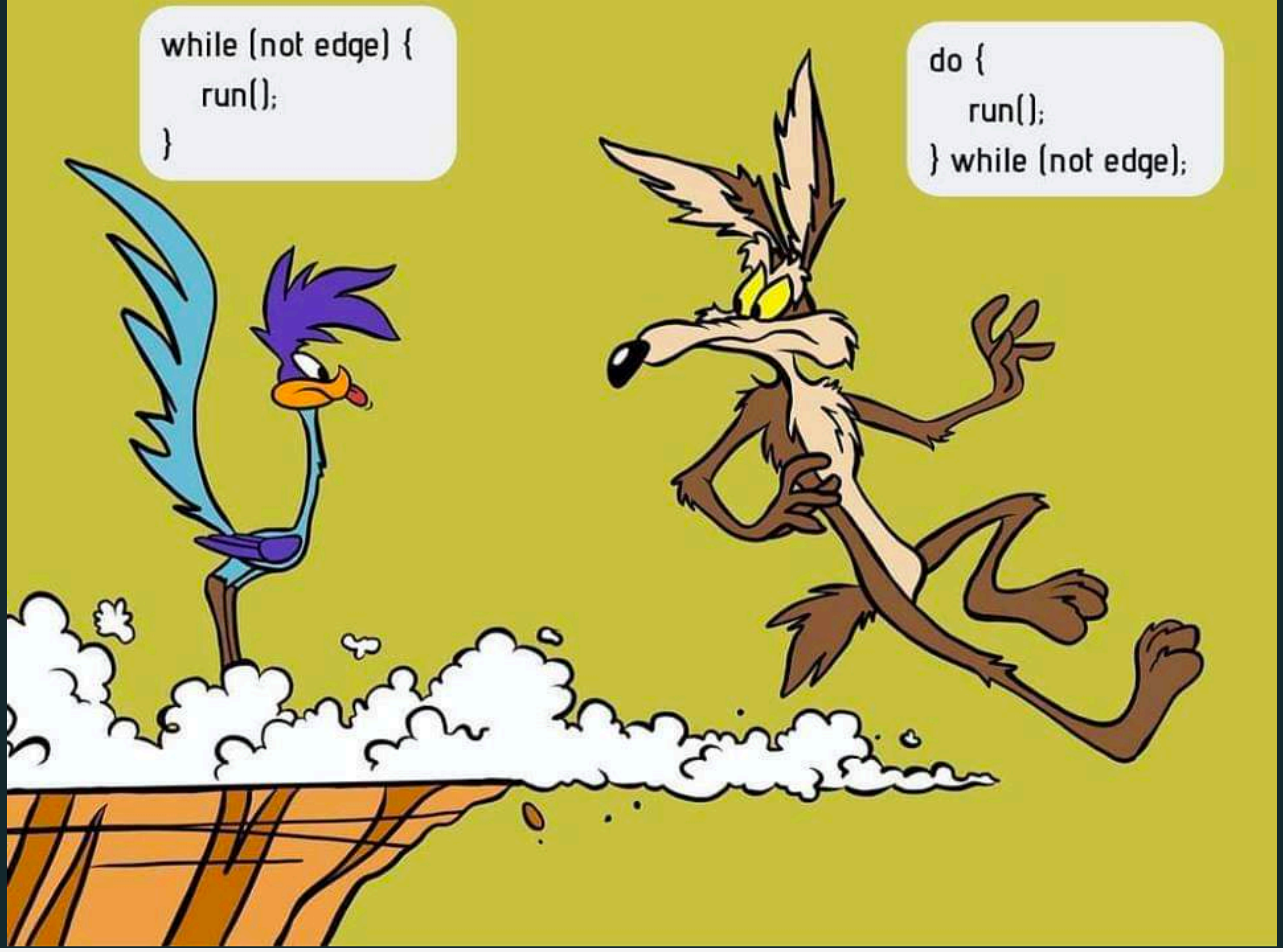
examples: average_numbers.cpp

Do-While Loops

A construct that repeatedly executes a block of sub-statements while the loop condition evaluates to true

- Do-While loops are ***post-test*** loops
- All Do-while loops execute at least 1 iteration

What is the final value of ***count***?



```
while (not edge) {
   run();
}
```

```
do {
   run();
} while (not edge);
```

```
int count = 0;
int num = 6;

do {
  num--;
  count++;
} while (num > 4);
```

Enumeration

An enumeration is a user-defined data type that is made up of integral constants (or ***enums***).

- Enumerated Types are excellent at representing ***state*** within a program!

```
enum MenuOption { ONE = 1, TWO, THREE, FOUR };

MenuOption state = ONE;

switch (state) {
  case ONE:
  case TWO:
  case THREE:
}
```

examples: bank.cpp

For Loops

for loops are commonly used when there is a ***finite*** number of iterations to be performed.

General Syntax:

```
for (initialExpression; conditionExpression; updateExpression) {
  // Loop body
}
```

- The **`initialExpression`** is executed one time before the block executes
- The **`conditionExpression`** defines the condition for executing the block
- The **`updateExpression`** is executed following every iteration of the block

examples: findMax.cpp

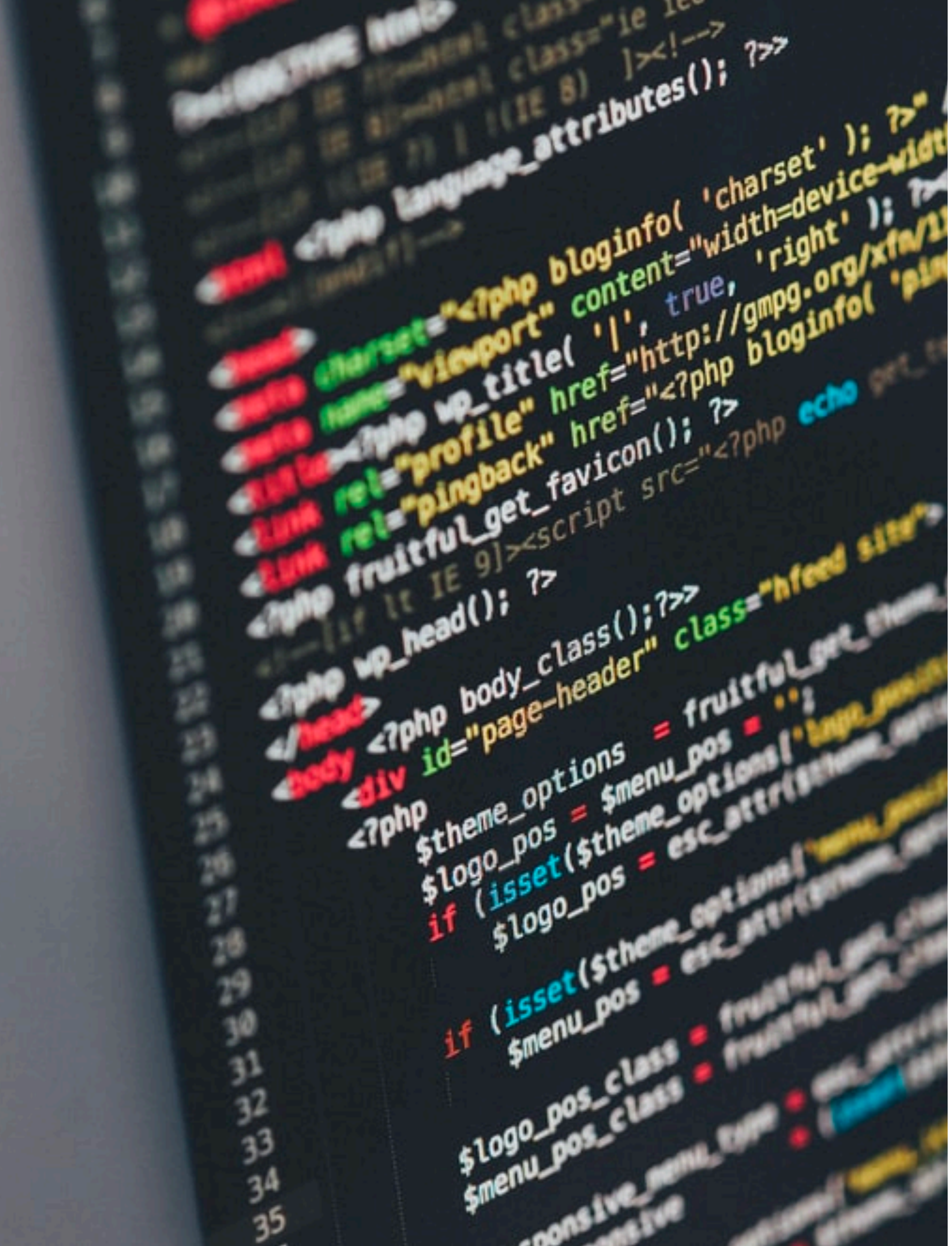
Nested Loops

- Nested for loops can be used to work in two dimensions
- The number of iterations in the nested for loop increase by a multiple of ***i * j * k * ... etc.***

```
int reps{0};

for (int i = 0; i < 10; i++) {
   for (int j = 0; j < 10; j++) {
      reps++;
   }
}

cout << reps; // what prints?
```

```
int reps{0};

for (int i = 0; i < 10; i++) {
   for (int j = 0; j < 10; j++) {
      for (int k = 0; k < 10; k++) {
         reps++;
      }
   }
}

cout << reps; // what prints?
```

examples: square.cpp



Challenge!

Lets build a program that outputs the following:

Prompt user for int: (user enters 5)

We draw this:

```
*****
*   *
*   *
*   *
*****
```

Characters, Strings, Cstrings

Common operations w/ strings

Character Operations

The ***cctype*** provides access to several character functions

- **isalpha(char) -** determines whether character is alpha character (a - z or A - Z)
- **isdigit(char) -** determines whether character is digit (0 - 9)
- **isspace(char) -** determines whether character is whitespace (i.e ` `, `\n`, etc)

NOTE: When comparing strings, we compare ascii values, so:

“Apple” != “apple”

C Strings

- NOT all character arrays are c strings. A ***valid*** c string MUST be null terminated
- No library is needed for using C strings
- ***#include <cstring>*** is a library that gives utility functions for manipulating c strings.

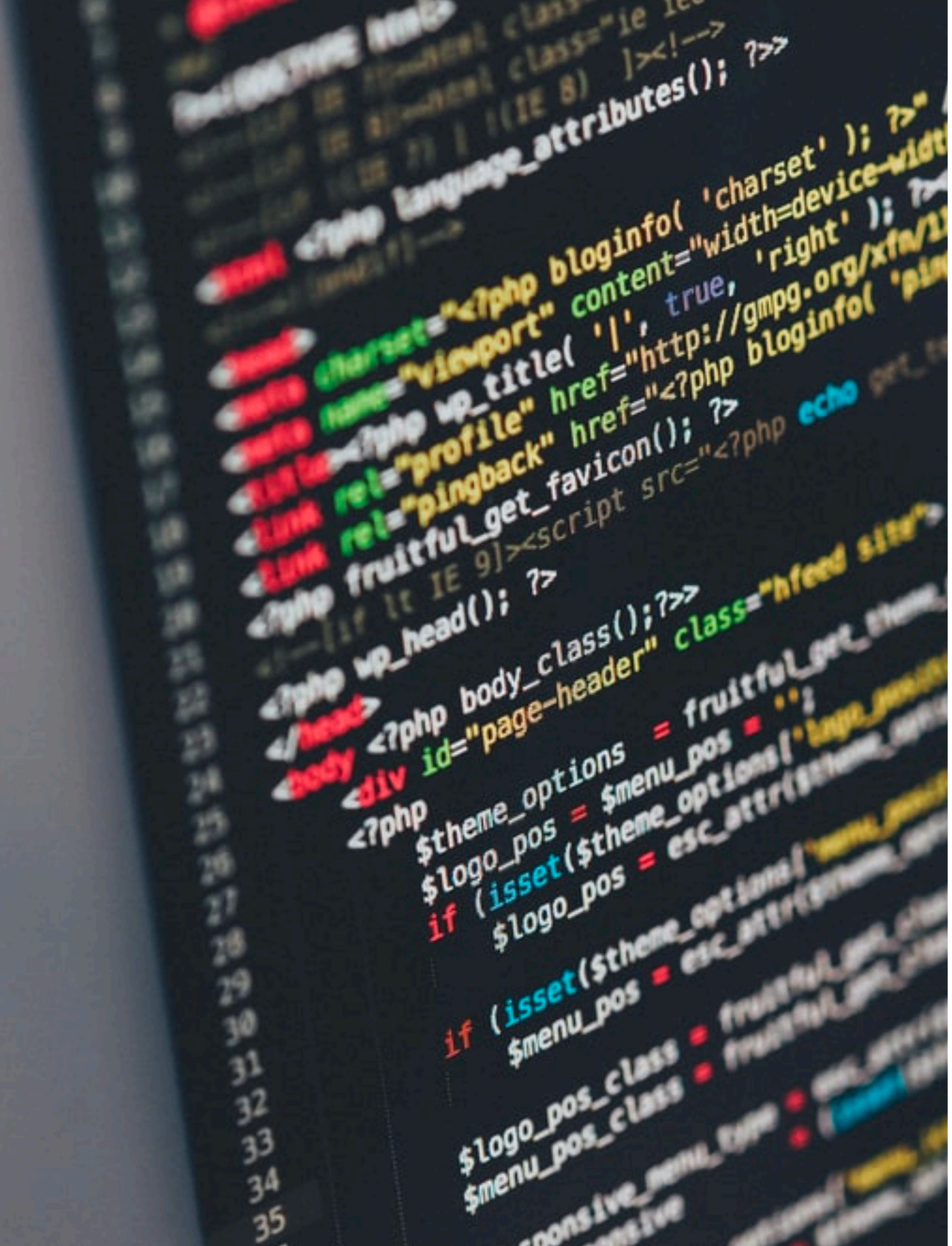
```
char s1[10]; // character array - can hold a C string

char s2[10] = { 'h', 'e', 'l', 'l', 'o', '\0'};

char s3[10] = "hello";

char s4[10] = ""; // empty c string with size 0
```

S3	h	e	l	l	o	\0				

Utility Functions for C strings

- ***strlen(cstring)* -** finds length of a cstring not including null character
- ***strcmp(s1, s2)* -** compares s1 to s2 for equality
- ***strcpy(s1, s2)* -** copy s2 into s1.
  - s1 must be large enough to contain the contents of s2. S2 can be a string literal OR another **`const char[]`**
  - If s2 is larger than s1, then the string will ***overflow*** the array.
- Remember with C strings we cannot use re-assignment. Must use strcpy.
- Access character of a cstring with `[]`

examples: strlen.cpp
strcmp.cpp



Challenge!

Write a program that given a cstring ***str[80],*** output the percentage of digits and letters present in the string.