8.x — Chapter 8 summary and quiz

Chapter Review

The specific sequence of statements that the CPU executes in a program is called the program's **execution path**. A **straight-line program** takes the same path every time it is run.

Control flow statements (also called **Flow control statements**) allow the programmer to change the normal path of execution. When a control flow statement causes the program to begin executing some non-sequential instruction sequence, this is called a **branch**.

A **conditional statement** is a statement that specifies whether some associated statement(s) should be executed or not.

If statements allow us to execute an associated statement based on whether some condition is `true`. **Else statements** execute if the associated condition is `false`. You can chain together multiple if and else statements.

A **dangling else** occurs when it is ambiguous which `if statement` an `else statement` is connected to. `Dangling else` statements are matched up with the last unmatched `if statement` in the same block. Thus, we trivially avoid `dangling else` statements by ensuring the body of an `if statement` is placed in a block.

A **null statement** is a statement that consists of just a semicolon. It does nothing, and is used when the language requires a statement to exist but the programmer does not need the statement to do anything.

Switch statements provide a cleaner and faster method for selecting between a number of matching items. Switch statements only work with integral types. **Case labels** are used to identify the values for the evaluated condition to match. The statements beneath a **default label** are executed if no matching case label can be found.

When execution flows from a statement underneath a label into statements underneath a subsequent label, this is called **fallthrough**. A `break statement` (or `return statement`) can be used to prevent fallthrough. The [[fallthrough]] attribute can be used to document intentional fallthrough.

Goto statements allow the program to jump to somewhere else in the code, either forward or backwards. These should generally be avoided, as they can create **spaghetti code**, which occurs when a program has a path of execution that resembles a bowl of spaghetti.

While loops allow the program to loop as long as a given condition evaluates to `true`. The condition is evaluated before the loop executes.

An **infinite loop** is a loop that has a condition that always evaluates to `true`. These loops will loop forever unless another control flow statement is used to stop them.

A **loop variable** (also called a **counter**) is an integer variable used to count how many times a loop has executed. Each execution of a loop is called an **iteration**.

Do while loops are similar to while loops, but the condition is evaluated after the loop executes instead of before.

For loops are the most used loop, and are ideal when you need to loop a specific number of times. An **off-by-one error** occurs when the loop iterates one too many or one too few times.

Break statements allow us to break out of a switch, while, do while, or for loop (also `range-based for loops`, which we haven't covered yet). **Continue statements** allow us to move immediately to the next loop iteration.

Halts allow us to terminate our program. **Normal termination** means the program has exited in an expected way (and the `status code` will indicate whether it succeeded or not). **std::exit()** is automatically called at the end of `main`, or it can be called explicitly to terminate the program. It does some cleanup, but does not cleanup any local variables, or unwind the call stack.

Abnormal termination occurs when the program encountered some kind of unexpected error and had to be shut down. **std::abort** can be called for an abnormal termination.

An **algorithm** is a finite sequence of instructions that can be followed to solve some problem or produce some useful result. An algorithm is considered to be **stateful** if it retains some information across calls. Conversely, a **stateless** algorithm does not store any information (and must be given all the information it needs to work with when it is called). When applied to algorithms, the term **state** refers to the current values held in stateful variables.

An algorithm is considered **deterministic** if for a given input (the value provided for `start`) it will always produce the same output sequence.

A **pseudo-random number generator (PRNG)** is an algorithm that generates a sequence of numbers whose properties simulate a sequence of random numbers. When a PRNG is instantiated, an initial value (or set of values) called a **random seed** (or **seed** for short) can be provided to initialize the state of the PRNG. When a PRNG has been initialized with a seed, we say it has been **seeded**. The size of the seed value can be smaller than the size of the state of the PRNG. When this happens, we say the PRNG has been **underseeded**. The length of the sequence before a PRNG begins to repeat itself is known as the **period**.

A **random number distribution** converts the output of a PRNG into some other distribution of numbers. A **uniform distribution** is a random number distribution that produces outputs between two numbers X and Y (inclusive) with equal probability.

Quiz time

Warning: The quizzes start getting harder from this point forward, but you can do it. Let's rock these quizzes!

Question #1

In lesson 4.x -- Chapter 4 summary and quiz, we wrote a program to simulate a ball falling off of a tower. Because we didn't have loops yet, the ball could only fall for 5 seconds.

Take the program below and modify it so that the ball falls for as many seconds as needed until it reaches the ground. Update the program to use all covered best practices (namespaces, constexpr, etc…).

```cpp
#include <iostream>

// Gets tower height from user and returns it
double getTowerHeight()
{
        std::cout << "Enter the height of the tower in meters: ";
        double towerHeight{};
        std::cin >> towerHeight;
        return towerHeight;
}

// Returns the current ball height after "seconds" seconds
double calculateBallHeight(double towerHeight, int seconds)
{
        const double gravity { 9.8 };

        // Using formula: s = (u * t) + (a * t^2) / 2
        // here u (initial velocity) = 0, so (u * t) = 0
        const double fallDistance { gravity * (seconds * seconds) / 2.0 };
        const double ballHeight { towerHeight - fallDistance };

        // If the ball would be under the ground, place it on the ground
        if (ballHeight < 0.0)
                return 0.0;

        return ballHeight;
}

// Prints ball height above ground
void printBallHeight(double ballHeight, int seconds)
{
        if (ballHeight > 0.0)
                std::cout << "At " << seconds << " seconds, the ball is at height: "
<< ballHeight << " meters\n";
        else
                std::cout << "At " << seconds << " seconds, the ball is on the
ground.\n";
}

// Calculates the current ball height and then prints it
// This is a helper function to make it easier to do this
void calculateAndPrintBallHeight(double towerHeight, int seconds)
{
        double ballHeight{ calculateBallHeight(towerHeight, seconds) };
        printBallHeight(ballHeight, seconds);
}

int main()
{
        double towerHeight{ getTowerHeight() };

        calculateAndPrintBallHeight(towerHeight, 0);
        calculateAndPrintBallHeight(towerHeight, 1);
        calculateAndPrintBallHeight(towerHeight, 2);
        calculateAndPrintBallHeight(towerHeight, 3);
        calculateAndPrintBallHeight(towerHeight, 4);
        calculateAndPrintBallHeight(towerHeight, 5);

        return 0;
}
```

Show Solution

Question #2

A prime number is a natural number greater than 1 that is evenly divisible (with no remainder) only by 1 and itself. Complete the following program by writing the `isPrime()` function using a for-loop. When successful, the program will print “Success!”.

```
// Make sure that assert triggers even if we compile in release mode
#undef NDEBUG

#include <cassert> // for assert
#include <iostream>

bool isPrime(int x)
{
    return false;
    // write this function using a for loop
}

int main()
{
    assert(!isPrime(0)); // terminate program if isPrime(0) is true
    assert(!isPrime(1));
    assert(isPrime(2));  // terminate program if isPrime(2) is false
    assert(isPrime(3));
    assert(!isPrime(4));
    assert(isPrime(5));
    assert(isPrime(7));
    assert(!isPrime(9));
    assert(isPrime(11));
    assert(isPrime(13));
    assert(!isPrime(15));
    assert(!isPrime(16));
    assert(isPrime(17));
    assert(isPrime(19));
    assert(isPrime(97));
    assert(!isPrime(99));
    assert(isPrime(13417));

    std::cout << "Success!\n";

    return 0;
}
```

Related content

`assert` is a preprocessor macro that terminates the program if the associated argument evaluates to false. So when we write `assert(!isPrime(0))`, we're meaning "if isPrime(0) is true, then terminate the program". We cover assert in more detail in lesson 9.6 -- Assert and static_assert.

Show Solution

Extra credit:

The for-loop in the above solution is suboptimal for two reasons:

- It checks even numbers. We know these aren't prime (except for 2).

- It checks every number up to `x` to see if it is a divisor. A non-prime number (a composite number) must have at least one divisor less than or equal to its square root, so checking for divisors beyond the square root of `x` is unnecessary. `std::sqrt(x)` (in the <cmath> header) returns the square root of `x`. While we could test all numbers up to `std::sqrt(x)`, calling `std::sqrt(x)` each iteration is slow. We can optimize `std::sqrt(x)` out of the comparison by squaring both sides of the comparison (h/t to reader JJag for suggesting this) (see the hint if you need additional help with this).

Show Hint

Update the above solution to implement both of these optimizations.

Show Solution

Question #3

Implement a game of Hi-Lo. First, your program should pick a random integer between 1 and 100. The user is given 7 tries to guess the number.

If the user does not guess the correct number, the program should tell them whether they guessed too high or too low. If the user guesses the right number, the program should tell them they won. If they run out of guesses, the program should tell them they lost, and what the correct number is. At the end of the game, the user should be asked if they want to play again. If the user doesn't enter 'y' or 'n', ask them again.

For this quiz, assume the user enters a valid number.

Use the Random.h header from 8.15 -- Global random numbers (Random.h).

Here's what your output should look like:

```
Let's play a game. I'm thinking of a number between 1 and 100. You have 7 tries to
guess what it is.
Guess #1: 64
Your guess is too high.
Guess #2: 32
Your guess is too low.
Guess #3: 54
Your guess is too high.
Guess #4: 51
Correct! You win!
Would you like to play again (y/n)? y
Let's play a game. I'm thinking of a number between 1 and 100. You have 7 tries to
guess what it is.
Guess #1: 64
Your guess is too high.
Guess #2: 32
Your guess is too low.
Guess #3: 54
Your guess is too high.
Guess #4: 51
Your guess is too high.
Guess #5: 36
Your guess is too low.
Guess #6: 45
Your guess is too low.
Guess #7: 48
Your guess is too low.
Sorry, you lose. The correct number was 49.
Would you like to play again (y/n)? q
Would you like to play again (y/n)? n
Thank you for playing.
```

Show Solution

We'll add error handling to this solution in lesson 9.x -- Chapter 9 summary and quiz.