

Ryan Baker

December 27, 2024

Contents

1	Functions	2
2	Scope	2
3	Conditions and Branches	3
3.1	Boolean Statements	3
3.2	if Statements	4
3.3	switch Statements	4
4	Loops	5
4.1	while Loops	5
4.2	for Loops	5
4.3	Control Flow Statements	5
4.3.1	break	5
4.3.2	continue	6
4.3.3	return	6

Lecture Objectives

1 Functions

- What is a function?
 - In programming, a function is a reusable block of code
 - It (optionally) takes input and (optionally) returns an output
- Why use functions?
 - We often want to repeat the same behavior on different pieces of data
 - Rather than pasting the same code many times, we use a function
 - * Functions help to keep code maintainable and readable
 - There is a balance to strike when extracting code into functions
 - * Too few functions results in long and repetitive code
 - * Too many functions will result in sub-optimal performance and a code base that is very hard to read
 - Every time a function is called a new frame needs to be pushed to the stack and we jump around the executable
- How to define a function: **type name(arguments)**
 - **type**: The return type of the function (can be `void`)
 - **name**: The function's name
 - **arguments**: The input arguments to a function
 - * Specified as **type name** in a comma separated list
 - * **Example**: `int add(int a, int b, int c) {...}`
 - Together, the function's name and arguments make up the signature
- How to call a function: **name(arguments)**
 - **Example**: `int sum = add(1, 2, 3); // sum = 6`

2 Scope

In C++, **scope** refers to the region of a program, where a variable or function is accessible and valid. It defines the part of the code where the name of an identifier is recognized.

- How is scope defined?
 - A scope is defined by curly braces: `{...}`
 - ... is said to be “within the scope” and everything else is outside

- Types of scope:
 - Global scope: Accessible from any part of the program
 - * Variables declared outside any function
 - Local scope: Accessible only within the same block (`{...}`)
 - * Variables declared within functions

3 Conditions and Branches

3.1 Boolean Statements

- What is a Boolean statement?
 - A `bool` type in C++ can either hold `true` or `false`
 - Therefore, a Boolean statement evaluates to `true` or `false`
 - They are often the inputs to conditional evaluation (for clear reasons)
- How to create a Boolean statement:
 - `bool(x)`: casts `x` to a type `bool`
 - * For numeric types, returns `true` if `x` \neq 0, `false` if `x` = 0
 - * `bool(1) == true; bool(-1) == true; bool(0) == false`
 - Comparison operators: return a boolean based on the comparison
 - * `==`, `!=`, `<`, `>`, `<=`, `>=` are all rather simple (`a == b` \leftrightarrow `a = b`)
 - Unary `!` operator: negation operator
 - * `!true == false` `!false == true`
 - * `!(5 == 6) == true` `!(0 == 0) == false`
 - Operator `&&`: and operator
 - * The **and** operator, `&&` returns true if both sides are true
 - * `(1 == 1) && (0 == 0) == true`
 - * `(1 > 0) && (0 > 1) == false`
 - Operator `||`: or operator
 - * The **or** operator, `||` returns true if either side is true
 - * `(1 == 1) || (1 == 0) == true`
 - * `(1 < 0) || (0 > 1) == false`

3.2 if Statements

- When to use an **if** statement:
 - Use **if** statements when you want to run a block of code conditionally
 - **Example:** You only want to log a user in **if** their password is correct
- **if** statement syntax: `if (condition) {...}`
 - Only executes the block of code (...) if the boolean expression is true
- Use of **else** to provide an alternative
 - `if (condition) {...} else {...}`
- Chaining **if** and **else** into **else if**
 - `if (cond 1) {...} else if (cond 2) {...} else {...}`
 - **else if** is not a new keyword, rather an **if** within an **else**
- The overhead of **if** statements
 - Every time an **if** statement is evaluated, the computer has to “jump” around the executable to the correct execution block
 - This entails an expensive load operation, making **if** statements relatively expensive
 - Overhead is minuscule, why even bother explaining this?
 - * It’s important to understand how things actually work
 - * Builds an understanding for the motivation of **constexpr**
 - * Sometimes you do actually care about nanoseconds-microseconds
 - For these reasons, programmers often use ternary operator
 - * `(condition) ? (if true) : (if false)`
 - * “returns” second argument if first argument (boolean) is true, else “returns” third argument
 - There are also often mathematical ways to get around using **if**

3.3 switch Statements

- Think of **switches** as **ifs** where you have an **int** condition, not a **bool**
- **switch** statement syntax: `switch(var) { cases... }`
 - The case corresponding to the **int** expression will be executed
 - If no case matches, **default** is executed
- The **break** statements are needed, else every case below will also execute
 - Sometimes this is desired behavior, but rarely
- Switches are often preferred to chained **if-else** statements when possible
 - They do not have the same overhead as many **ifs**

4 Loops

Often times, we need a certain block of code to run multiple times. Rather than copying and pasting said block, we can use a loop. A loop is a structure that allows a block of code to be run a specified amount of times.

4.1 while Loops

- When to use a **while** loop?
 - You have a block that you need to run **while** a condition is true
 - **Example: while** the player is playing the game, render the screen
- **while** loop syntax: **while** (condition) {...}
 - ... will run **while** condition is true
- When to use a **do while** loop?
 - You need the block of code to run at least one time
 - **do** {...} **while** (condition)
 - * The **do** keyword's only purpose is to pair the **while** loop to a scope written before it, rather than after
 - A **do while** loop runs the block first, and checks the condition second

4.2 for Loops

- When to use a **for** loop?
 - You need a block of code to run a set number of times
 - * **for** loops are much more flexible than **while** loops
 - * Any **while** loop can be a **for** loop
- **for** loop syntax: **for** (init; condition; update) {...}
 - **init** is a piece of code that will run once before the loop runs
 - **condition** works just like a **while** loop
 - **update** runs at the end of every loop iteration
 - You can leave any or all fields empty
 - * The default behavior for an empty condition is “**true**”

4.3 Control Flow Statements

4.3.1 break

- **break** is used to “break out” of loops and switches early
- Can only be used within a loop or a switch, not any scope

4.3.2 `continue`

- `continue` is used to “continue” to the next iteration of a loop
- Can only be used within a loop structure

4.3.3 `return`

- `return` is used to “return” a value from a function