C++ Control Flow Essentials

Ryan Baker

December 31, 2024

Contents

Lecture Objectives

By the end of this lecture, you should:

- Know how, and when, to use functions in C++
- Be able to utilize control structures to write logical programs
- Understand the notion of "scope", and how it affects visibility

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 `<datatype 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`

```cpp
// function.cpp
#include <iostream>

int add(int a, int b, int c)
{
    return a + b + c;
}

int main()
{
    std::cout << add(2, 4, 5) << std::endl;
    std::cout << add(5, 1, 0) << std::endl;
    std::cout << add(6, 0, 9) << std::endl;
    return 0;
}
```

2 Scope

In C++, **scope** refers to the region of a program, where a variable or function is accessible.

- How is scope defined?
  - A scope is defined by curly braces: $\{\dots\}$
  - ... is “within scope” and everything else is “out of scope”
- 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 ($\{\dots\}$)
    - Variables declared within functions

```cpp
// scope.cpp
#include <iostream>

int global_int = 10;
bool global_bool = 3;

int foo()
{
    int foo_int = 4;
    return foo_int;
}

int main()
{
    {
        // this is a narrower scope
        int narrow_int = 3;
    }
    // can no longer see "narrow_int"

    int local_int = 5;
    std::cout << local_int << std::endl;

    return 0;
}
```

3 Conditions and Branches

Often times in programming, we need to perform different actions depending on some condition. For example, *if* the user already has an account, *then* show them the login screen. *Else*, show them the account creation screen. Conditions and branches help us achieve behavior like this.

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: NOT operator
    - `!true == false`   `!false == true`
    - `!(5 == 6) == true`   `!(0 == 0) == false`
  - Operator `&&`: AND operator
    - The **and** operator, `&&` returns true if both operands are true
    - `(1 == 1) && (0 == 0) == true`
    - `(1 > 0) && (0 > 1) == false`
  - Operator `||`: OR operator
    - The **or** operator, `||` returns true if either operand is true
    - `(1 == 1) || (1 == 0) == true`
    - `(1 < 0) || (0 > 1) == false`

```
// boolean.cpp
#include <iostream>

int main()
{
    std::cout << (1 == 1)    << std::endl; //  true or 1
    std::cout << (4 == 3)    << std::endl; // false or 0
    std::cout << bool(-7)    << std::endl; //  true or 1
    std::cout << (!(1 == 1)) << std::endl; // false or 0
    std::cout << (-1 < 1)    << std::endl; //  true or 1
    std::cout << (!(!(true))) << std::endl; // true or 1

    return 0;
}
```

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 (<expression>) {...}`
  - Only executes the block of code (...) if the boolean expression is true
- Use of `else` to provide an alternative:
  - `if (<expression>) {...} else {...}`

```
// ifelse.cpp
#include <iostream>

int main()
{
    int n = 45;
    if (n > 30)
        std::cout << "n is greater than 30" << std::endl;
    else
        std::cout << "n is 30 or less" << std::endl;
    return 0;
}
```

- Chaining `if` and `else` into `else if`
  - `if (<expr1>) {...} else if (<expr2>) {...} else {...}`
  - `else if` is not a new keyword, rather an `if` within an `else`

```
// elseif.cpp
int main()
{
    int n = 45;
    if (n > 60)      { /* do something */ }
    else if (n > 30) { /* do something */ }
    else if (n > 10) { /* do something */ }
    else             { /* do something */ }
    return 0;
}
```

- 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
  - For these reasons, programmers often use ternary operator

3.3 Ternary Operator

- Ternary syntax: `(<expression>) ?  (<if true>) :  (<if false>)`
- “returns” second argument (`<if true>`) if first argument (`<expression>`) is true, else “returns” third argument (`<if false>`)

```
// ternary.cpp
#include <iostream>

int foo(int x)
{
    return (x > 5 ? 10 : 0); // standard use of ternary
}

int main()
{
    std::cout << foo(6) << std::endl; // 10
    std::cout << foo(1) << std::endl; // 0

    // non-standard use of ternary
    (2 > 3 ? std::cout << "true" : std::cout << "false");
    std::cout << std::endl;

    return 0;
}
```

- Ternary operators can be nested, and even used to run lines of code

3.4 `switch` Statements

- Think of `switch`es as `if`s 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

```cpp
// switch.cpp
#include <iostream>

int main()
{
    int today = 5;
    switch (today)
    {
        case 0:
            std::cout << "Monday" << std::endl; break;
        case 1:
            std::cout << "Tuesday" << std::endl; break;
        case 2:
            std::cout << "Wednesday" << std::endl; break;
        case 3:
            std::cout << "Thursday" << std::endl; break;
        case 4:
            std::cout << "Friday" << std::endl; break;
        case 5:
            std::cout << "Saturday" << std::endl; break;
        case 6:
            std::cout << "Sunday" << std::endl; break;
        default:
            std::cout << "Invalid day." << std::endl;
    }
    return 0;
}
```

- 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 `if`s
  - They also enhance readability and maintainability of code bases

4 Loops

Often times, we need a certain block of code to run multiple times. Rather than copying and pasting the block, we can use a loop.

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

```cpp
// while.cpp
#include <iostream>

int main()
{
    int x = 0;
    while (x < 5)
    {
        std::cout << x << std::endl;
        x++;
    }
    return 0;
}
```

- 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

```cpp
// dowhile.cpp
#include <iostream>

int main()
{
    int x = 0;
    do // do-while loops run before checking the condition
    {
        std::cout << x << std::endl;
    } while (x != 0);
    return 0;
}
```

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 rewritten as 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`”

```
// for.cpp
#include <iostream>

int main()
{
    for (int i = 0; i < 5; ++i)
    {
        std::cout << i << std::endl;
    }
    return 0;
}
```

5 Control Flow Statements

5.1 `break` Keyword

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

```
// break.cpp
#include <iostream>

int main()
{
    for (int i = 0; i < 10; ++i)
    {
        if (i >= 5)
            break; // use "break" to exit loop early
        std::cout << i << std::endl;
    }

    return 0;
}
```

5.2 `continue` Keyword

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

```
// break.cpp
#include <iostream>

int main()
{
    for (int i = 0; i < 10; ++i)
    {
        if (i >= 5)
            break; // use "break" to exit loop early
        std::cout << i << std::endl;
    }

    return 0;
}
```

5.3 return Keyword

- `return` is used to “return” a value from a function
- Can only be used within a function

```
// return.cpp
#include <iostream>

void foo(int x)
{
    if (x > 10)
        return; // return early
    std::cout << x << std::endl;
    // implicit return statement
}

int bar(int x)
{
    if (x > 10)
        return 0; // return early
    return x;
}

int main()
{
    return 0; // return 0 signifies all went well
}
```

5.4 goto Keyword

- `goto` is used to jump to a label (defined as `<name>:`)
- Use of this keyword is heavily discouraged by many people

```
// goto.cpp
#include <iostream>

int main()
{
    int x = 10;
Label:
    if (x > 5)
    {
        std::cout << (x--) << std::endl;
        goto Label;
    }
    return 0;
}
```