

# C++ Control Flow

Ryan Baker

January 15, 2025

## Contents

<b>1</b>	<b>Functions</b>	<b>2</b>
1.1	Passing by Value vs. Passing by Reference . . . . .	2
1.2	Function Overloading . . . . .	3
<b>2</b>	<b>Scope</b>	<b>3</b>
2.1	Types of Scope . . . . .	3
2.1.1	Global Scope . . . . .	3
2.1.2	Local Scope . . . . .	3
2.1.3	Anonymous Scope . . . . .	3
2.2	Namespaces . . . . .	4
2.2.1	Namespace Operator :: . . . . .	4
2.2.2	using Namespaces . . . . .	4
<b>3</b>	<b>Conditions and Branching</b>	<b>4</b>
3.1	Boolean Expressions . . . . .	4
3.1.1	bool() Casts . . . . .	4
3.1.2	Comparison Operators: ==, !=, <, <=, >, >= . . . . .	4
3.1.3	Logical Operators: !, &&,    . . . . .	5
3.2	if Statements . . . . .	5
3.2.1	The Overhead of if Statements . . . . .	5
3.3	switch Statements . . . . .	5
3.4	Ternary Operator ? : . . . . .	5
<b>4</b>	<b>Loops</b>	<b>5</b>
4.1	while Loops . . . . .	5
4.1.1	do while Loops . . . . .	5
4.2	for Loops . . . . .	5
4.2.1	Blank Fields . . . . .	5
<b>5</b>	<b>Control Flow Keywords</b>	<b>5</b>
5.1	break Keyword . . . . .	5
5.2	continue Keyword . . . . .	5
5.3	return Keyword . . . . .	5

# 1 Functions

Often, we want to simplify complex problems and repeat computations, and functions help by breaking tasks into smaller units. This makes the code easier to reuse, maintain, and test.

- **Definition:** A function in C++ is a named block of code that takes inputs, does something with them, and can return a result.

- **Declaring a Function:** `type name(args) {...}`

`type`: The return type of the function (e.g., `int`, `void`, etc.)

`name`: The function's name

`args`: The function arguments or parameters as a comma separated list

```
int add(int a, int b, int c);
```

- `void` is used as the return type for functions that don't return anything:

```
void greet();
```

- **Functions and the Stack:** Every time a function is called, its local variables and arguments are pushed atop the stack.

## 1.1 Passing by Value vs. Passing by Reference

Arguments that are passed into a function are, by default, copied. This means that the original value will remain unmodified:

```
1 #include <iostream>
2
3 void increment(int x) { x++; }
4
5 int main()
6 {
7     int x = 0;
8     increment(x); // copy of x is passed to
        increment
9     std::cout << x << std::endl; // x = 0
10 }
```

Very often, this is the intended effect. One other consideration is that for very large objects copying is expensive. The alternative is to pass arguments by reference:

```

1 #include <iostream>
2
3 void increment(int& x) { x++; }
4
5 int main()
6 {
7     int x = 0;
8     increment(x); // reference to x is passed
9     std::cout << x << std::endl; // x = 1
10 }

```

## 1.2 Function Overloading

Together, a function's name and arguments make up its *signature*. Because the arguments are included, two functions with the same name but different arguments will not cause duplicate symbol errors:

```

1 // two functions can have the same name
2 int add(int a, int b)          { return a + b; }
3 int add(int a, int b, int c) { return a + b + c; }

```

Note that it is not enough for the functions to have differently names arguments.

## 2 Scope

Scope is defined by {} in C++. A scope is a region of the program where a varibale or symbol is valid.

### 2.1 Types of Scope

#### 2.1.1 Global Scope

Variables in global scope are accessible throughout the program.

#### 2.1.2 Local Scope

Variables in local, or function scope, are accessible only in their function.

#### 2.1.3 Anonymous Scope

Scopes can be defined anonymously, helping with naming conflicts:

```

1 int main()
2 {

```

```

3     { // anonymous scope
4         int x = 0;
5     }
6 }

```

## 2.2 Namespaces

Namespaces are a tool that helps us deal with naming conflicts. Consider you're writing a library with a function called `init()`. You'd be bound to end up with a naming conflict, which is where namespaces help.

```

1 namespace Library
2 {
3     void init();
4 }

```

### 2.2.1 Namespace Operator ::

```
Library::init();
```

### 2.2.2 using Namespaces

```
using namespace std;

using std::cout;
```

## 3 Conditions and Branching

Often times in programming, we want to choose which path to execute based on certain conditions.

### 3.1 Boolean Expressions

A boolean variable can represent true or false. Thus, a boolean expression is an expression that evaluates to true or false.

#### 3.1.1 bool() Casts

Evaluates to true if not 0, false if 0 for numeric types.

#### 3.1.2 Comparison Operators: ==, !=, <, <=, >, >=

```
==, !=, <, <=, >, >=
```

### 3.1.3 Logical Operators: `!`, `&&`, `||`

`!`, `&&`, `||`

## 3.2 `if` Statements

`if` Statements are used to execute a block of code if a condition is true:

### 3.2.1 The Overhead of `if` Statements

After the evaluation of an `if` statement, the execution jumps around and loads new instructions.

## 3.3 `switch` Statements

Like `if` statements but for integer expressions not boolean.

## 3.4 Ternary Operator `?` `:`

# 4 Loops

### 4.1 `while` Loops

#### 4.1.1 `do while` Loops

### 4.2 `for` Loops

#### 4.2.1 Blank Fields

# 5 Control Flow Keywords

### 5.1 `break` Keyword

### 5.2 `continue` Keyword

### 5.3 `return` Keyword