C++ Programming Basics

Introduction to Modern C++

Ryan Baker

February 15, 2025

Lecture Objectives

- To understand how and why we use functions in C++.
- To understand the notion of scope and how it relates to variable lifetimes and visibility.
- To leverage conditions and branching to alter program flow.
- To understand the semantics of various branching constructs.
- To become familiar with looping constructs in C++.

Contents

1 Functions

What is a Function? A function is a reusable block of code that performs a specified task. Instead of repeating code, functions allow us to write it once and call it many times. In this way, functions help make programs more organized and readable.

Consider the following code snippet:

```cpp
#include <iostream>

int main()
{
    int x = 25, y = 4;
    int q = x / y;
    int r = x % y;

    std::cout << x << " / " << y << " = "  << q
              << " remainder " << r << std::endl;
}
```

This snippet calculates and prints the quotient and remainder associated with dividing two integers x and y. If we decide that we'd like to replicate this logic on many pairs of x and y, our code can become messy quite quickly:

```cpp
#include <iostream>

int main()
{
    int x = 25, y = 4;
    int q = x / y;
    int r = x % y;
    std::cout << x << " / " << y << " = "  << q
              << " remainder " << r << std::endl;

    x = 40;
    y = 17;
    q = x / y;
    r = x % y;
    std::cout << x << " / " << y << " = "  << q
              << " remainder " << r << std::endl;

    x = 234;
    y = 123;
    q = x / y;
21     r = x % y;
22     std::cout << x << " / " << y << " = "  << q
23               << " remainder " << r << std::endl;
24 }
```

Above, notice that lines 6-10, lines 13-16, and lines 20-23 contain identical logic. This makes that logic an ideal candidate to be placed into a function.

1.1 Defining Functions

Building upon the motivation for a function described above, we will define a function called `print_divmod()`.

The first thing to consider when defining a function in C++ the datatype that this function will `return`, its *return type*. For example, a function that adds two integers will likely return an `int`. Because `print_divmod()` only performs a calculation and prints the result, its return type is `void`.

Note: To *return* some piece of data from a function is to pass that piece of data back to the function caller.

```
// write the function's return type before its name
// print_divmod returns 'no type'
void print_divmod();
```

`print_divmod()` needs to accept as input two integers: x and y. In C++, function input arguments are a comma separated list of `<type> <name>`:

```
// print_divmod takes two integers called x and y
void print_divmod(int x, int y);
```

The above is a valid *function declaration*. Recall that variable initialization can be split into declaration and definition. To *define* a function is to write its body:

```
void print_divmod(int x, int y)
{
    int q = x / y;
    int r = x % y;
    std::cout << x << " / " << y << " = "  << q
              << " remainder " << r << std::endl;
}
```

1.2 Calling Functions

To call `print_divmod()`, we use the following syntax:

```cpp
// ... print_divmod() defined above

int main()
{
    print_divmod(25, 4);    // x = 25,  y = 4
    print_divmod(40, 17);   // x = 40,  y = 14
    print_divmod(234, 123); // x = 234, y = 123
}
```

Note that in C++, `()` is the function call operator. It is used to invoke functions and callable objects in C++, such as `print_divmod()`.

If the function has a `return` type, its output can be assigned to a variable of the same type:

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

int main()
{
    int sum = add(21, 21); // sum = 42
}
```

1.3 Function Arguments

By default in C++, arguments are passed as *copies* into functions. This means that the memory occupied by the variable passed into a function is **not** the same memory occupied by the variable that the function receives. Instead, a copy of the variable is made and passed.

```cpp
#include <iostream>

void foo(int x)
{
    x = 10; // reassign x to 10
}

int main()
{
    int x = 5;
    foo(x);
    std::cout << x << std::endl; // 5
}
```

As a brief aside, we can use the address-of operator `&` to verify that both variables do indeed occupy different memory:

```cpp
#include <iostream>

void foo(int x)
{
    // print the address of x
    std::cout << &x << std::endl;
}

int main()
{
    int x = 5;

    // print the address of x
    std::cout << &x << std::endl;

    foo(x);
}
```

1.4 Function Signatures

A foundational concept relating to functions in C/C++ is a *function signature*. Every function has a signature. A function signature is made up of the function's **name** and **argument types** (and it's surrounding namespace), and it serves as a unique identifier by which the compiler can refer to the function.

Because the function argument types are a part of its signature, we can define two functions with the same name but different arguments:

```cpp
int add(int x, int y) { return x + y; }
float add(float x, float y) { return x + y; }
int add(int x, int y, int z) { return x + y + z; }
```

The clumsy nature of the above code provides motivation for *templates*.

2 Scope

A variable's *scope* is the region of a program where its declaration is visible. Each declaration within a C++ program occurs within a scope. Each program has a *global scope*, which contains the entire program. Narrower scopes are typically denoted by curly braces `{}` in C++.

```
#include <iostream> // <------------------------+
                    //                          |
void foo()          //  <---------------+       |
{                   //              foo |       |
// do foo things    //            scope |       |
}                   //  <---------------+       |
                    //                          |
int main()          //  <---------------+       |
{                   //                  |       |
    {               //   <---------+    |    global
        int x;      //     scope A |    |     scope
    }               //   <---------+    |       |
                    //                 main     |
    {               //   <---------+  scope     |
        int x;      //     scope B |    |       |
    }               //   <---------+    |       |
                    //                  |       |
    return 0;       //                  |       |
}                   //  <---------------+       |
                    // <------------------------+
```

Note that in the above program, x is declared twice within `main()`. This, however, doesn't cause problems because the declarations occur in disjoint scopes. In general, you can access symbols in your current scope or any enclosing scope.

2.1 Global Scope

Global scope refers to the scope outside of any class, function, or namespace. Variables declared in global scope are called *global variables* and can be accessed from any part of the program. Global variables are stored in *static memory* and their lifetime is the entire duration of the program.

```
#include <iostream>

int g = 10; // globally scoped variable

int main() {}
```

2.2 Local Scope

Local scope refers to the region within a *block* of code defined by curly braces `{}`. This scope limits the variable's visibility and lifetime to that specific block.

```cpp
#include <iostream>

int g = 30; // global variable

void some_function()
{
    int l = 42; // local variable
    std::cout << g << std::endl;
    std::cout << l << std::endl;
}

int main()
{
    g = 10;
    // l = 15; error, cannot access 'l'
    some_function();
}
```

2.3 namespace Scope

Namespaces provide a method for preventing naming conflicts. Entities declared within a `namespace` block are placed in a *namespace scope*, which prevents them from being mistaken for identically named entities in other scopes.

```cpp
int x = 1;

namespace MyNamespace
{
    int x = 2; // different variable than on line 1
}
```

2.3.1 Scope Resolution Operator ::

The scope resolution operator `::` is used to access namespace-scoped variables and functions:

```cpp
std::cout << x << std::endl;                // 1
std::cout << MyNamespace::x << std::endl; // 2
```

Notice that `cout` and `endl` are prefixed with `std::`. Indeed, this is because they reside in the standard (std) `namespace`.

Also notice that, when on line 1, we access `x` with no namespace prefix, the compiler smartly infers that we mean to access the global `x`. If, however, we had a third `x` that was local:

```cpp
#include <iostream>

int x = 1;

namespace MyNamespace { int x = 2; }

int main()
{
    int x = 3; // seems I forgot 25 other letters

    std::cout << x << std::endl;
    std::cout << MyNamespace::x << std::endl;
}
```

Here the compiler fairly assumes we are refering to the local variable `x`. Output:

```
3
2
```

The compiler assumes you are refering to the variable with the narrowest scope:

```cpp
#include <iostream>

int x = 0;

int main()
{
    int x = 1;
    {
        int x = 2;
        {
            int x = 3;
            std::cout << x << std::endl;
        }
        std::cout << x << std::endl;
    }
    std::cout << x << std::endl;
}
```

Output:

```
3
2
1
```

To explicitly reference a variable in global namespace, use the scope resolution operator `::` with no namespace:

```cpp
#include <iostream>

int x = 1;

int main()
{
    int x = 2;
    std::cout << ::x << std::endl; // 1
}
```

2.3.2 `using` Namespaces

In C++, the `using` keyword allows us to simplify access `namespace` members. Instead of using the resoluton operator `::` every time we refer to a namespace member, we can bring the entire `namespace` or specific elements into the current scope.

The most common way to use a `namespace` is with the `using namespace` directive. This allows all elements of the specified namespace to be accessed without needing the `namespace::` prefix:

```cpp
#include <iostream>

using namespace std; // now we don't need std::

int main()
{
    cout << "Hello, World!" << endl;
}
```

Instead of importing an entire namespace, we can selectivley bring in elements with `using` declarations:

```cpp
#include <iostream>

```

```cpp
using std::cout, std::endl;

int main()
{
    cout << "Hello, World!" << endl;
}
```

This approach is generally prefered over `using namespace std;` because it reduces the risk of naming conflicts.

3 Conditions and Branching

Often times, the desired behavior of our program depends on some piece of data that is known only at runtime. For example, we may only want to display some piece of information to a user only **if** that user is authenticated. The act of decision-making for control flow is known as *branching*.

3.1 Boolean Expressions

A Boolean expression is an expression that evaluates to `true` or `false`. These expressions are fundamental in decision-making structures like `if` statements and `switch` statements.

The simplest Boolean expressions are the keyword literals `true` and `false`.

```cpp
std::cout << true << std::endl;  // 1
std::cout << false << std::endl; // 0
```

3.1.1 `bool()` Casts

All numerical types in C++ can be cast into a `bool` type. Generally, if the number is 0 it is converted to `false`, else it is `true`.

```cpp
std::cout << bool(1) << std::endl;  // 1
std::cout << bool(0) << std::endl;  // 0
std::cout << bool(-1) << std::endl; // 1
```

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

Comparison operators return `true` or `false` based on the relationship between two values:

Operator	Meaning	Example
`==`	Equal to	`x == y`
`!=`	Not equal to	`x != y`
`<`	Less than	`x < y`
`<=`	Less than or equal to	`x <= y`
`>`	Greater than	`x > y`
`>=`	Greater than or equal to	`x >= y`

```
std::cout << (1 > 0) << std::endl;  // 1
std::cout << (1 == 0) << std::endl; // 0
std::cout << (0 == 0) << std::endl; // 1
```

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

Logical operators allow combining multiple Boolean expressions:

Operator	Meaning	Example
`!`	Logical NOT	`!(x)`
`&&`	Logical AND	`(x > 0) && (x < 10)`
`||`	Logical OR	`(x < 0) || (x > 10)`

```
std::cout << (1 && 0) << std::endl; // 0
std::cout << (1 || 0) << std::endl; // 1
std::cout << (!(1)) << std::endl;   // 0
```

Short-Circuiting The logical operators `&&` and `||` will terminate evaluation early if the result of the expression can be decided early. This both allows for better performance and allows the programmer to simplify what would otherwise be very bulky constructs.

```
#include <iostream>

bool f() {
    std::cout << "f" << std::endl;
    return true;
}

bool g() {
    std::cout << "g" << std::endl;
    return false;
}

```

```cpp
bool h() {
    std::cout << "h" << std::endl;
    return true;
}

int main()
{
    f() && g() && h();
    f() || g() || h();
}
```

Output:

```
f
g
f
```

3.2 `if` Statements

The `if` statement allows executing a block of code *if* a given condition is `true`:

```cpp
#include <iostream>

int main()
{
    int num;
    std::cin >> num;

    if (num > 0)
    {
        std::cout << "positive" << std::endl;
    }
}
```

The `else` block executes when the condition evaluates to false:

```cpp
#include <iostream>

int main()
{
    int num;
    std::cin >> num;

    if (num > 0)
    {
        std::cout << "positive" << std::endl;
    }
    else
    {
        std::cout << "non-positive" << std::endl;
    }
}
```

As a stylistic choice, for one-line `if` statements, you may omit the curly braces `{}` and the single line will serve as a block:

```
#include <iostream>

int main()
{
    int num;
    std::cin >> num;

    if (num > 0)
        std::cout << "positive" << std::endl;
    else
        std::cout << "non-positive" << std::endl;
}
```

To check multiple conditions, we use `else if`:

```
#include <iostream>

int main()
{
    int num;
    std::cin >> num;

    if (num > 0)
        std::cout << "positive" << std::endl;
    else if (num < 0)
        std::cout << "negative" << std::endl;
    else
        std::cout << "zero" << std::endl;
}
```

Note that `else if` is not a keyword, rather it is the natural conclusion of C++ allowing one statement to follow an `if` statement without curly braces `{}`.

3.3 switch Statements

The switch statement is used when multiple conditions depend on a single **integer** variable. It provides an alternative to multiple if else if statements that is often both cleaner and more performant.

```
#include <iostream>

int main()
{
    int day;
    std::cin >> day;

    switch (num)
    {
        case 0:
            std::cout << "Monday" << std::endl;
            break;
        case 1:
            std::cout << "Tuesday" << std::endl;
            break;
        // ...
        case 6:
            std::cout << "Sunday" << std::endl;
            break;
        default:
            std::cout << "Invalid day" <<
    std::endl;
    }
}
```

3.3.1 Switch break Statement

Each case should (usually) end with a break; to prevent fall-through. Without a break, execution continues into the next case:

```
int x = 1;

switch (x) // switch without breaks
{
    case 0:
        std::cout << "zero" << std::endl;
    case 1:
        std::cout << "one" << std::endl;
    case 2:
            std::cout << "two" << std::endl;
        case 3:
            std::cout << "three" << std::endl;
        default:
            std::cout << "invalid" << std::endl;
}
```

Output:

```
one
two
three
invalid
```

Had we included `break` statements, the output would have been as expected:

```
one
```

The `default` label is a special label reserved for `switch` statements in C++. It gets executed when none of the other cases are evaluate to `true`.

```
int x = 1;

switch (x)
{
    case 0:
        break;
    default:
        std::cout << "default case executed" <<
    std::endl;
}
```

3.3.2 How `switch` Works

It would be a mistake to assume that a `switch` is merely a stylistic preference over a series of `if else`'s. In fact, the way that the two constructs achieve branching behavior is very different.

In a series of `if else` statements, each one will be evaluated, then upon a hit the corresponding block will be entered. In a `switch` statement, a *jump table* is set up allowing control to be transfered directly to the correct `case`. This is why the variable within the `switch` condition must be an integer type.

3.4 Ternary Operator ? :

The ternary operator, or the conditional operator, is a concise way to execute one of two expressions based on a condition.

As you may guess from the name *ternary*, it takes three arguments e_1, e_2, e_3 in the form `e1 ? e2 : e3`. If e_1 is `true`, then it returns e_2, else it returns e_3. Hence, the following `if` statement:

```
if (x % 2 == 0)
    x = x / 2;
else
    x = 3 * x + 1;
```

can be rewritten as:

```
x = ((x % 2 == 0) ? (x / 2) : (3 * x + 1));
```

Note that in the above, all of the parentheses `()` are merely a stylistic choice intended to help the reader easily parse the expression.

4 Loops

Loops allow a program to execute a block of code multiple times. C++ has two main types of loops: the `while` loop and the `for` loop.

4.1 `while` Loops

A `while` loop runs **while** the given condition remains `true`.

```
while (condition)
{
    // do cool things (or maybe boring things)
}
```

One common application of a `while` loop is validating user input:

```
#include <iostream>

int main()
{
    int x;
    std::cout << "Enter a positive number: \n";
    std::cin >> x;

    // while the user refuses to comply
    while (!(x > 0))
    {
        std::cin >> x;
    }

    std::cout << "x = " << x << std::endl;
}
```

4.1.1 `do {} while` Loops

A `do {} while` loop is similar to a `while` loop, but it evaluates the condition at the **end** of the loop body, rather than the beginnning. This guarantees that the loop will execute at least once, even in the initial condition is `false`.

```
do
{
    // something fun
} while (condition);
```

The input validation program can be refactored using a `do {} while` loop:

```
#include <iostream>

int main()
{
    int x;
    std::cout << "Enter a positive number: \n";

    do
    {
        std::cin >> x;
    } while (!(x > 0));

    std::cout << "x = " << x << std::endl;
}
```

4.2 `for` Loops

A `for` loop is commonly used when the number of iterations is either known beforehand or can be calculated. It consists of three parts:

```
for (initialization; condition; update)
```

```
{
    // do something awesome
}
```

- **Initialization**: Piece of code that is run once before the loop begins.
- **Condition**: The loop runs as long as this condition is `true`.
- **Update**: A piece of code that is run at the end of each iteration.

4.3 Control Flow Keywords

4.3.1 break Keyword

The `break` statement within a loop is used to exit the loop immediately.

```
for (int i = 0; i < 5; ++i)
{
    if (i == 3) // dastardly number 3
        break;
    std::cout << i << std::endl;
}
```

Output:

```
0
1
2
```

4.3.2 continue Keyword

The `continue` statement is used to skip the rest of the current iteration and move on to the next one.

```
for (int i = 0; i < 5; ++i)
{
    if (i == 3) // abominable number 3
        continue;
    std::cout << i << std::endl;
}
```

Output:

```
0
```

```
1
2
4
```