Contents

1 Structure of a C++ Program

1.1 Section Overview

This section introduces the foundational structure and components of a basic C++ program. Understanding these elements is crucial before diving into more complex logic. The key components covered include:

- **Keywords:** Reserved words native to C++.
- **Identifiers:** Programmer-defined names for variables, functions, and objects.
- **Operators:** Symbols that perform specific mathematical or logical operations.
- **Punctuation:** Structural characters like semicolons (`;`) and curly braces ({}).
- **Syntax:** The grammatical rules combining all the above elements.
- **Pre-processor Directives:** Instructions for the compiler, starting with a hashtag (`#`).
- **Functions & Namespaces:** The `main` function and the standard namespace (`std`).
- **Basic I/O:** Using `cin` and `cout` for console input and output.

1.2 Overview of the Structure of a C++ Program

Keywords Keywords form the built-in vocabulary of the C++ language. Because they are strictly reserved by the compiler, you cannot redefine their meaning or use them as names for your own variables. C++ contains approximately 90 keywords—significantly more than languages like Java (50) or Python (33)—which reflects its highly complex grammar. You do not need to memorize this list upfront; you will naturally learn them as you use them.

Examples of keywords include `int` and `return`.
Note: Elements like `#include`, `main`, `cin`, and `cout` are **not** keywords.

Identifiers Identifiers are names created by the programmer to represent entities like variables, functions, or classes. Unlike keywords, identifiers should be meaningful to the humans reading the code. For example, `favorite_number` is a custom identifier. Names pulled from the standard library, such as `cin` and `cout`, also function as identifiers in your code.

Operators Operators perform specific operations on data. Beyond standard arithmetic operators (`+`, `-`, `*`, `/`), C++ relies heavily on unique operators for input, output, and structure:

- `<<` : **Stream Insertion Operator**. Inserts data from the right side into the output stream on the left (commonly used with `cout`).
- `>>` : **Stream Extraction Operator**. Extracts data from the input stream on the left and stores it into the variable on the right (commonly used with `cin`).
- `::` : **Scope Resolution Operator**. Specifies the context or namespace of an identifier (e.g., `std::cout`).

Punctuation and Syntax Punctuation in C++ provides structure. It includes semicolons (`;`) to terminate statements, curly braces ({}) to define blocks of code or scope, parentheses (`()`) for function arguments, and quotes (`""`) for strings.

Syntax is the synthesis of keywords, identifiers, operators, and punctuation according to the strict rules of C++. The compiler relies entirely on correct syntax to translate your human-readable code into machine code; it executes exactly what is written and will not guess your intentions.

Code Example: Bringing It Together Here is a basic program demonstrating the distinct elements discussed above:

```cpp
// Pre-processor directive (Not a keyword)
#include <iostream>

// Using the standard namespace to avoid typing std:: everywhere
using namespace std;

// 'int' is a keyword, 'main' is an identifier
int main() {
    // 'int' is a keyword, 'favorite_number' is a programmer-defined identifier
    int favorite_number;

    // 'cout' is an identifier, '<<' is the stream insertion operator
    // ";" is punctuation ending the statement
    cout << "Enter your favorite number: ";

    // 'cin' is an identifier, '>>' is the stream extraction operator
    cin >> favorite_number;

    cout << "Amazing! " << favorite_number << " is my favorite too!" << endl;

    // 'return' is a keyword
    return 0;
}
```

1.3 #include Preprocessor Directive

What is the C++ Pre-processor? The C++ pre-processor is a program that processes your source code *before* the compiler sees it. Its primary jobs are to format the file and execute specific instructions to prepare the code for compilation.

First, the pre-processor strips all the comments from the source file and replaces each comment with a single space. Then, it looks for and executes pre-processor directives.

Pre-processor Directives Pre-processor directives are lines in the source code that begin with a pound or hashtag symbol (`#`). While there are many available directives, the most commonly used is the `#include` directive.

When the pre-processor encounters an `#include` directive, it essentially copies the contents of the referenced file and pastes it directly into the current source file, recursively processing any directives within that new file as well. By the time the compiler actually sees the source code, all comments have been removed and all pre-processor directives have been resolved and replaced with the necessary code.

Conditional Compilation Pre-processor directives are commonly used to conditionally compile code. For instance, you might want to compile a specific portion of your source code only if the program is running on a Windows operating system. You would use a pre-processor directive to check the OS; if it is Windows, it includes Windows-specific libraries. If it is macOS, it might include macOS libraries or use the `#error` directive to abort the compilation with a specific message.

An Important Distinction It is critical to understand that **the C++ pre-processor does not understand C++ syntax**. It does not read or interpret the logic of your program. It simply follows the explicit pre-processor directives to get the source code ready. The compiler is the program that actually understands and translates the C++ code.

1.4 Comments

What are Comments? Comments are programmer-readable explanations, notes, or annotations directly inside the source code that add context or meaning to what the program is doing.

Because the pre-processor strips all comments out and replaces them with a single space, **comments never make it to the compiler**. They are entirely for human benefit—either for yourself when you return to the code later, or for other programmers who need to read and maintain your work.

Types of Comments in C++ C++ supports two styles of comments:

- **Single-Line Comments:** Initiated with two forward slashes (`//`). Everything following the slashes on that specific line is ignored. These can be placed on their own line or at the end of a line of code.

- **Multi-Line Comments:** Initiated with a forward slash and an asterisk (`/*`) and terminated with an asterisk and a forward slash (`*/`). Everything between these two markers is ignored, regardless of how many lines it spans. These are completely free-form and can be indented or styled however you prefer.

```cpp
/* This is a multi-line comment.
   It is often used at the top of files for:
   Author: Frank
   Date: 11/11/2017
   License Information
*/

#include <iostream>

int main() {
    int favorite_number; // This is a single-line comment
    return 0;
}
```

Best Practices for Commenting The overarching philosophy in modern programming is "literate programming," meaning your code should be as self-documenting and readable as possible.

- **Don't comment the obvious:** If you have a line of code like `return 0;` or `a + b`, you do not need a comment saying "returning zero" or "adding a and b". If the code is simple (like prompting a user for a number and printing it), it likely doesn't need comments.
- **Explain the "Why" and the "Complex":** Use comments when the code is highly complicated, uses a complex algorithm, or relies on a clever efficiency tweak that makes the logic less obvious. For example, noting that you are "using a modified version of Dijkstra's algorithm to improve space efficiency" is highly valuable context for the next programmer.
- **Keep style consistent:** Don't unnecessarily mix comment styles within the same block of logic. Pick a style and stick with it. Multi-line comments are generally preferred for large header blocks at the top of a file (detailing author, dates, and copyright info), while single-line comments are used throughout the body of the code.
- **Do not use comments for version control:** Avoid keeping a running log of who changed what and when (e.g., "Frank fixed a bug on 11/11"). This is unreliable because not every programmer will adhere to it. Use a dedicated version control system like Git instead.
- **Update comments when you update code:** If you modify a piece of code, you must update the corresponding comment. A comment that describes an old, replaced algorithm is worse than no comment at all, as it will actively mislead the person reading it.

1.5 The main() function

Execution and Return Values Every C++ program must have exactly one `main` function somewhere. Even if a program consists of hundreds of files, one of them must contain the `main` function. Note that the name `main` must strictly be written in lowercase letters.

When a C++ program executes, the `main` function is called by the operating system, and the code between its curly braces executes. When execution hits the `return` statement, the program returns an integer value to the operating system. If the return value is zero, it indicates that the program terminated successfully. If the return value is not zero, the operating system can check this value to determine what went wrong.

Two Versions of main There are two valid versions of `main` per the C++ specification.

The first version returns an integer and has empty parentheses. It expects no information from the operating system to run. This is the version primarily used in this course.

```cpp
int main() {
    return 0;
}
```

The second version expects information passed from the operating system, which is very common for command-line applications. In this case, we tell the compiler we need two pieces of information:

- `argc` (Argument Count): The number of pieces of information passed in.
- `argv` (Argument Vector): The actual arguments passed into the program, represented as an array of strings (e.g., `program.exe`, `argument1`, `argument2`).

```
int main(int argc, char *argv[]) {
    return 0;
}
```

What is a Function? `main` happens to be a special example of a function. A function is basically a name used to reference a block of code (everything between the curly braces). As programs become more complex, we write our own functions and classes to better modularize and organize the code.

1.6 Namespaces

Naming Conflicts As C++ programs become more complex, they combine your own code, the C++ standard library code, and third-party developer libraries. Eventually, you will encounter a naming conflict—for example, a situation where a third-party library and the standard library both define an entity named `cout`. The C++ compiler will not know which one to use.

The Scope Resolution Operator C++ allows developers to use namespaces as containers to group their code entities. For example, if you create a namespace called `Frank` and define `cout` inside it, a programmer would use `Frank::cout` to access it. If they want the standard library version, they use `std::cout`.

The double colon (`::`) is called the scope resolution operator. It resolves exactly which name we want to use, drastically reducing the possibility of naming conflicts.

Three Ways to Handle the Standard Namespace Many programmers find it tedious to repeatedly type `std::` for `cin`, `cout`, and `endl`. C++ provides a few mechanisms to handle this:

1. Explicit Specification This is the safest method. You explicitly tell the compiler to use the standard namespace via the scope resolution operator every time.

```
#include <iostream>

int main() {
    std::cout << "Hello" << std::endl;
    return 0;
}
```

2. The using namespace Directive This directive tells the compiler to bring in the entire standard namespace. You no longer need to type `std::`. This reduces code clutter and is great for teaching, but it is not the best solution for large programs because it brings in *all* names from that namespace, reintroducing the risk of naming conflicts.

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello" << endl;
    return 0;
}
```

3. Qualified using Directives This variant tells the compiler exactly which specific names you want to use from a namespace. It allows you to write code without the scope resolution operator for those specific entities, without pulling in any other unwanted names from the standard library. This is considered a better practice for larger programs.

```cpp
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main() {
    cout << "Hello" << endl;
    return 0;
}
```

1.7 Basic Input and Output (I/O) using cin and cout

Stream Abstraction C++ uses a stream abstraction to handle Input/Output (I/O) operations on devices like the console and keyboard. To use the standard I/O streams, you must include the `<iostream>` header. The standard defines four main streams:

- `cout`: An output stream that defaults to the console/screen.
- `cin`: An input stream that defaults to the keyboard.
- `cerr`: An output stream for standard errors.
- `clog`: An output stream for standard logging.

Output with cout and the Insertion Operator The insertion operator (`<<`) is used with output streams. It inserts the value of the operand on its right into the output stream on its left.

```cpp
cout << "Hello World";
```

Because C++ uses stream abstraction, you can chain multiple insertions in the same statement, making basic I/O very straightforward.

```cpp
cout << "Hello " << "World" << endl;
```

It is important to note that the insertion operator does *not* automatically add line breaks. You must explicitly move the cursor to the next line using either the `endl` stream manipulator or the newline character (`\n`). The `endl` manipulator will also flush the stream buffer, ensuring that the data is immediately written to the console, whereas `\n` only advances to the next line.

Input with cin and the Extraction Operator The extraction operator (`>>`) extracts information from the `cin` input stream and stores it into the variable to its right. The way the extracted information is interpreted depends entirely on the data type of that variable. If the variable is an `int`, `cin` will attempt to read an integer; if it is a `double`, it will look for a real number.

```cpp
int num1;
cout << "Enter an integer: ";
cin >> num1;
```

Just like insertion, extraction operators can be chained to read multiple variables in a single statement.

```cpp
int num1, num2;
cout << "Enter two integers separated by a space: ";
cin >> num1 >> num2;
```

How the Input Buffer Works When a user types on the keyboard, the characters are not immediately read by `cin`. They are stored in a buffer. `cin` only begins processing this buffer when the user presses the **Enter** key.

Furthermore, `cin` extraction uses **whitespace** (spaces, tabs, newlines) as terminating boundaries for the value being extracted. All leading whitespace is completely ignored.

Buffer Behavior Examples Understanding the buffer is critical to avoiding unexpected behavior:

- **Data Left in the Buffer:** If a program asks for two separate integers in sequence, but the user types `100 200` and presses enter at the very first prompt, `cin` extracts `100` and assigns it to the first variable. However, `200` remains in the buffer. When the program reaches the next `cin` statement, it will not wait for the user to type anything; it immediately extracts the `200` that was left behind in the buffer.
- **Type Mismatches:** If a program expects an `int` but the user types `10.5`, `cin` will read the `10` as a valid integer and assign it. The decimal point is not a valid integer character, so `cin` stops extracting, leaving `.5` in the buffer. If the program subsequently asks for a `double`, it will instantly read that leftover `0.5`.
- **Failed State:** If a program expects an `int` and the user types a string like `Frank`, `cin` encounters the 'F', realizes it cannot make an integer, and enters a "fail state". The input operation fails, the variable will hold an undetermined value (or zero), and all subsequent `cin` operations will be unreliable until the stream is cleared. In real-world applications, programmers often read all input as strings first and then safely convert them to numbers to avoid crashing the input stream.

```cpp
#include <iostream>
using namespace std;

int main() {
    int num1;
    double num3;

    cout << "Enter an integer: ";
    cin >> num1;

    cout << "Enter a double: ";
    cin >> num3;

    cout << "The integer is: " << num1 << endl;
    cout << "The double is: " << num3 << endl;

    return 0;
}
```

2 Variables and Constants

2.1 Section overview

In this section of the course, we will explore the fundamental concepts of variables and constants, which are essential building blocks for virtually every C++ program you will write.

The key topics covered in this section include:

- **Variables:** What they are, how to declare them, rules for naming them, and best practices for styling.
- **Primitive Types:** The core data types built into C++, including integers, floating-point numbers, booleans, and characters.
- **The sizeof Operator:** How to determine the exact amount of memory storage allocated to different variables.
- **Constants:** What constants are, how to declare and use them, and why they are critical for keeping your code correct and easy to modify.
- **Literals and Constant Expressions:** Understanding fixed values embedded directly in your code.

2.2 What is a variable?

Variables and Computer Memory To understand variables, it helps to be familiar with the basic architecture of a typical computer system, which consists of a CPU, Random Access Memory (RAM), and a bus that moves data between them.

RAM is a contiguous block of storage used by the computer to hold information, including both computer instructions and data. You can think of RAM as a massive grid of individual memory cells, where each cell has its own specific, numerical memory location (or address).

Binding and Abstraction If programmers had to write code using these exact hardware memory locations—for example, instructing the computer to "move the value 21 to memory location 1002"—programming would be incredibly tedious and highly prone to errors. Furthermore, every time the program ran, the operating system might assign a different physical memory address, breaking the hardcoded logic.

Instead of dealing with raw memory addresses, C++ (like most modern languages) allows you to associate a human-readable name with a memory location. This process is called **binding**. When you bind the name `age` to a memory location, you can simply tell the compiler to "move 21 to `age`". You no longer need to know or care about the actual physical address the hardware is using.

In this way, a variable is an **abstraction** for a memory location. And, as the name implies, the contents of a variable can vary; if a person has a birthday, you can easily change the value stored in `age` from 21 to 22.

Declaration and Static Typing In C++, variables have two primary properties: a **name** and a **type**.

C++ is a **statically typed** language. This means the compiler enforces data types at compile time rather than when the program is actually executing. Therefore, the compiler must explicitly know what type of data is legally allowed to be stored in a variable before you ever try to use it.

If you try to assign a value to a variable without first declaring its type, the compiler will throw an error. You must always declare a variable before using it.

```
int main() {
    // THIS WILL CAUSE A COMPILER ERROR
    // The compiler does not know what 'age' is or what it can hold.
    age = 21;

    // THIS IS CORRECT
    // We declare 'age' as an integer first.
    int age;
    age = 21;

    // The value can vary later in the program
    age = 22;

    return 0;
}
```

2.3 Declaring and Initializing Variables

Variable Declaration Syntax The syntax for declaring a variable in C++ is straightforward: first, specify the type, then the name, and finally terminate the statement with a semicolon.

Common primitive types include:

- `int`: An integer or whole number.
- `double`: A floating-point (real) number.
- `string`: A sequence of characters.

C++ also allows you to declare variables of non-built-in types. Through Object-Oriented Programming, you can create your own custom types (like an `Account` or `Person`) and declare variables for them using the exact same syntax.

Rules for Naming Variables C++ enforces strict rules for variable names:

- They can contain letters, numbers, and underscores.
- The first character **cannot** be a number (it must be a letter or underscore).
- You cannot use C++ reserved keywords (e.g., `int`, `return`).
- You cannot declare a variable name that has already been declared in the same scope.
- Variable names cannot contain spaces or special mathematical characters (e.g., `$`, `+`).
- C++ is **case-sensitive**. `Age`, `age`, and `AGE` are all treated as completely different variables.

Note on `cout`: The word `cout` technically conforms to naming rules. However, once you include `<iostream>`, the standard library has already defined it. You cannot redefine it in that scope.

Naming Style and Best Practices Beyond the compiler's strict rules, you must also consider style and readability. The most important rule of styling is **consistency**.

- **Word Separation:** Choose a convention to separate words in long variable names. The two most common are **CamelCase** (capitalizing the first letter of each subsequent word, e.g., `roomWidth`) and **snake_case** (using underscores, e.g., `room_width`). This course will primarily use the underscore style.
- **Meaningful Names:** Never use obscure abbreviations. Instead of `moe`, write `mass_of_earth`. It takes slightly longer to type but makes the code infinitely more maintainable.
- **Declare Close to Use:** Do not declare a massive list of variables at the very top of your file. Best practice is to declare a variable immediately before you actually need to use it.

Initializing Variables A variable declaration without an initial value is called **uninitialized**. This is incredibly dangerous in C++ and is a common source of bugs. When you declare an uninitialized variable, C++ allocates a memory location for it but does not clean it out. The variable will contain whatever random sequence of ones and zeros happened to be left in that memory cell by a previous program (often referred to as a "garbage value").

You must always initialize your variables. C++ provides three ways to do this:

- **Assignment Operator:** `int age = 21;`
- **Constructor Initialization:** `int age (21);`
- **C++11 List Initialization:** `int age {21};` (This is the most highly recommended style in modern C++ as it enforces stricter safety checks across different types).

Code Example: Calculating Room Area Here is a live code example demonstrating declaring, initializing, and using variables close to their first use point.

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << "Enter the width of the room in feet: ";

    // Declaring and initializing right before use
    int room_width {0};
    cin >> room_width;

    cout << "Enter the length of the room in feet: ";

    // Declaring and initializing right before use
    int room_length {0};
    cin >> room_length;

    cout << "The area of the room is " << (room_width * room_length) << " square feet." <<
    endl;

    return 0;
}
```

2.4 Global Variables

Local vs. Global Scope Up until now, we have declared variables within the curly braces of the `main` function. These are called **local variables** because their scope (or visibility) is limited entirely to the block of code in which they are declared. This is generally the safest and recommended way to program.

However, variables can also be declared outside of any function. These are called **global variables**.

- Global variables can be accessed and modified from anywhere in your entire program.
- Unlike local variables (which contain garbage values if uninitialized), global variables are automatically initialized to zero by the compiler.

The Problem with Global Variables While global variables might seem convenient, they come with significant risks. Because they can be changed from anywhere, debugging a program with thousands of lines of code becomes incredibly difficult if a global variable's state is unexpectedly modified. In modern C++, relying on global variables is generally discouraged.

Variable Shadowing It is possible to have a global variable and a local variable with the exact same name. When the compiler encounters that variable name, it always looks locally first. If it finds a local declaration, it will use the local variable and ignore the global one. This is known as **shadowing** the outer variable.

```cpp
#include <iostream>
using namespace std;

// This is a GLOBAL variable
int age {16};

int main() {
    // This is a LOCAL variable.
    // It "shadows" the global 'age' variable within main().
    int age {18};

    // This will print 18, as the compiler looks locally first.
    cout << age << endl;

    return 0;
}
```

2.5 C++ Built-in Primitive Types

Fundamental Types and Memory Primitive types (or fundamental data types) are implemented directly by the C++ language. It is crucial to understand that, unlike some other languages, the exact size and precision of these types depend heavily on the compiler and the specific platform architecture you are using. You can check these limits by including the `<climits>` header.

Memory sizes are expressed in bits. The formula to calculate how many distinct values can be stored in a given number of bits is 2^n (where n is the number of bits). For example, an 8-bit allocation can hold 256 distinct values.

Character Types Used to represent single characters (like 'A' or 'x').

- `char`: Typically implemented as 8 bits. It easily supports the standard Latin character set.
- *Note:* Characters in C++ are enclosed in **single quotes** (e.g., `'J'`). Double quotes represent strings.

Integer Types Used to represent whole numbers. By default, all integers in C++ are **signed** (meaning they can hold positive and negative values). If you only want to store zero and positive values, you must explicitly use the `unsigned` keyword.

- `short`: Small integers.
- `int`: Standard integers (perfectly acceptable for most basic applications).
- `long`: Large integers.
- `long long`: Exceptionally large integers.

Floating-Point Types Used to represent real numbers with decimal points (e.g., 3.14159). Because computers have finite memory, floating-point numbers are stored internally as approximations using a mantissa and an exponent (similar to scientific notation).

- `float`: Standard precision.
- `double`: Double precision (used most commonly).
- `long double`: Used for extremely large or microscopically small numbers.

Boolean Type The `bool` data type represents true/false states. In C++, `0` is evaluated as false, and any non-zero value is true. When printed to the console, `false` outputs as `0` and `true` outputs as `1`.

List Initialization and Overflows If you attempt to calculate or store a number that is too large for its assigned data type, an **overflow** occurs. With traditional assignment (e.g., `short product = 30000 * 1000;`), the compiler will wrap the number around into a garbage negative value without failing, leading to severe logical errors.

This is why **C++11 List Initialization** (using curly braces {}) is so highly recommended. If you try to initialize a variable with a value that doesn't fit (a narrowing conversion), list initialization will catch it and generate a compiler error immediately.

Code Example: Primitive Types

```cpp
#include <iostream>
using namespace std;

int main() {
    // Character (Single quotes!)
    char middle_initial {'J'};

    // Unsigned Integer (Positive only)
    unsigned short exam_score {55};

    // Long Long Integer with C++14 digit separators (tick marks)
    long long people_on_earth {7600000000};

    // Floating Point Types
    double pi {3.14159};
    long double huge_amount {2.7e120}; // Scientific notation

    // Boolean
    bool game_over {false};

    // Outputting to console
    cout << "Initial: " << middle_initial << endl;
    cout << "Earth Pop: " << people_on_earth << endl;
    cout << "Game Over flag (prints 0): " << game_over << endl;

    /* OVERFLOW EXAMPLE
    short val1 {30000};
    short val2 {1000};
    short product {val1 * val2}; // Logical Error: Result is too big for a short!
    */

    return 0;
}
```

2.6 What is the Size of a Variable (sizeof)

The sizeof Operator C++ has a built-in operator called `sizeof` that returns the number of bytes used to represent any specific data type or variable in memory.

To determine the size of a specific data type, you place the name of the type inside parentheses (e.g., `sizeof(int)` or `sizeof(double)`). This tells you the exact storage size in bytes on the specific machine and compiler you are currently using.

You can also use `sizeof` to determine the size of specific variables. When using it with variable names, the parentheses are optional. Both `sizeof(age)` and `sizeof age` are completely valid syntax.

Header Files: ¡climits¿ and ¡cfloat¿ The `sizeof` operator derives its integral type information from the `<climits>` include file, and its floating-point information from `<cfloat>`.

These include files also provide a collection of extremely useful defined constants that allow you to determine the precision and limits of primitive types on your system. For example:

- `INT_MAX` / `INT_MIN`: The maximum and minimum values you can store in an integer.
- `CHAR_MAX` / `CHAR_MIN`: The maximum and minimum values for a character.

Why Size Matters in C++ You might wonder why you need to worry about the specific byte sizes of data types. In higher-level languages like Java or Python, you are far removed from the hardware and rarely need to think about memory allocation. However, C++ operates at a lower level, much closer to the machine. Understanding the size and precision of your types is critical—especially when dealing with arrays, structures, and preventing the types of integer overflows discussed in the previous section.

Code Example: Using sizeof and climits Below is a demonstration of how to output type sizes, variable sizes, and system limits. Note that the output values (like 4 bytes for an integer) are typical for a 32-bit compiler on a 64-bit operating system, but may vary depending on your specific hardware and compiler.

```cpp
#include <iostream>
#include <climits> // Required for limits like INT_MAX

using namespace std;

int main() {
    // 1. Using sizeof with primitive types
    cout << "sizeof information for types:" << endl;
    cout << "char: " << sizeof(char) << " bytes." << endl;
    cout << "int: " << sizeof(int) << " bytes." << endl;
    cout << "unsigned int: " << sizeof(unsigned int) << " bytes." << endl;
    cout << "short: " << sizeof(short) << " bytes." << endl;
    cout << "long: " << sizeof(long) << " bytes." << endl;
    cout << "long long: " << sizeof(long long) << " bytes." << endl;

    cout << "float: " << sizeof(float) << " bytes." << endl;
    cout << "double: " << sizeof(double) << " bytes." << endl;
    cout << "long double: " << sizeof(long double) << " bytes." << endl;

    // 2. Using climits to find minimum and maximum values
    cout << "\nMinimum values:" << endl;
    cout << "char: " << CHAR_MIN << endl;
    cout << "int: " << INT_MIN << endl;
    cout << "short: " << SHRT_MIN << endl;

    cout << "\nMaximum values:" << endl;
    cout << "char: " << CHAR_MAX << endl;
    cout << "int: " << INT_MAX << endl;
    cout << "short: " << SHRT_MAX << endl;

    // 3. Using sizeof with variable names
    int age {21};
    double wage {22.24};

    cout << "\nsizeof information for variables:" << endl;
    // Syntax variant 1: with parentheses
    cout << "age is " << sizeof(age) << " bytes." << endl;
    // Syntax variant 2: without parentheses
    cout << "age is " << sizeof age << " bytes." << endl;
    cout << "wage is " << sizeof(wage) << " bytes." << endl;

    return 0;
}
```

```
sizeof information for types:
char: 1 bytes.
int: 4 bytes.
unsigned int: 4 bytes.
short: 2 bytes.
long: 4 bytes.
long long: 8 bytes.
float: 4 bytes.
double: 8 bytes.
long double: 12 bytes.

Minimum values:
char: -128
int: -2147483648
short: -32768

Maximum values:
char: 127
int: 2147483647
short: 32767
```

```
sizeof information for variables:
age is 4 bytes.
age is 4 bytes.
wage is 8 bytes.
```

2.7 What is a constant?

Variables vs. Constants Constants are very similar to C++ variables: they have names, follow the same naming rules, occupy memory storage, and are usually typed.

The critical difference is that **the value of a C++ constant cannot change once it is declared**. If you attempt to assign a new value to a constant later in your code, the compiler will immediately throw a "read-only" error.

Why use Constants? Constants make your code significantly more readable and maintainable. Imagine a program that calculates payroll and frequently uses the number `12` for the months in a year. If you hardcode `12` everywhere, another programmer reading your code might not immediately know what `12` represents.

Furthermore, if you need to update a value (e.g., changing a tax rate from `0.06` to `0.07`), doing a "Find and Replace" on the literal number across thousands of lines of code is dangerous—you might accidentally replace a `0.06` that represents something entirely different. By defining a constant named `sales_tax` at the top of your file, you only have to update the value in one single place.

Types of Constants in C++ C++ provides several ways to create and use constants:

1. Literal Constants Literals are values directly written into your code. Examples include `12`, `1.56`, `"Frank"`, and the character `'J'`. You can explicitly define the type of an integer or floating-point literal by appending specific suffixes:

- `U` or `u`: Unsigned integer (e.g., `12U`)
- `L` or `l`: Long integer or Long double (e.g., `12L` or `1.56L`)
- `F` or `f`: Float (e.g., `1.56f`)

Escape Codes: Character literal constants preceded by a backslash are called escape codes. They are commonly embedded in string literals to format console output.

- `\n`: Newline
- `\t`: Tab

2. Declared Constants (The `const` Keyword) This is by far the most common and recommended way to declare constants in modern C++. The syntax is identical to declaring a variable, but you precede it with the `const` keyword. **You must initialize a declared constant immediately;** otherwise, the compiler will generate an error.

```
1 const double pi {3.14159};
2 const int months_in_year {12};
```

3. Defined Constants (Preprocessor Directives) You will often see defined constants in older, legacy C/C++ code using the `#define` preprocessor directive.

```
1 #define pi 3.1415926
```

When the preprocessor runs, it simply executes a blind "find and replace," swapping the word `pi` with the number before the code reaches the compiler. **Do not use this method in modern C++.** Because the preprocessor does not understand C++ syntax, it cannot perform type-checking, leading to hard-to-find bugs.

2.8 Declaring and Using Constants

Refactoring and Pseudocode Before writing code, programmers often use **pseudocode**—a natural language outline of the algorithm needed to solve a problem. Once the logic is sound, they write the actual code.

Refactoring is the process of improving the structure and readability of existing code *without* changing its external behavior.

Code Example: Carpet Cleaning Estimator Below is a live code example demonstrating how to refactor a program that initially relied on literal constants into a robust program using declared constants.

By declaring `price_per_room`, `sales_tax`, and `estimate_expiry` as constants at the top of the program, we ensure the numbers cannot be accidentally altered during execution. If the business ever raises its prices or the tax rate changes, the programmer only needs to update the single constant declaration, rather than hunting through the code for literal values.

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << "Hello, welcome to Frank's Carpet Cleaning Service" << endl;
    cout << "\nHow many rooms would you like cleaned? ";

    int number_of_rooms {0};
    cin >> number_of_rooms;

    // Declared Constants (Refactored from hardcoded literals)
    const double price_per_room {32.50};
    const double sales_tax {0.06};
    const int estimate_expiry {30}; // days

    cout << "\nEstimate for carpet cleaning service" << endl;
    cout << "Number of rooms: " << number_of_rooms << endl;
    cout << "Price per room: $" << price_per_room << endl;

    // Note: In future lessons, these inline calculations will be
    // moved into their own variables or functions to avoid code duplication!
    cout << "Cost: $" << price_per_room * number_of_rooms << endl;
    cout << "Tax: $" << price_per_room * number_of_rooms * sales_tax << endl;

    cout << "==============================" << endl;
    cout << "Total estimate: $"
         << (price_per_room * number_of_rooms) + (price_per_room * number_of_rooms * sales_tax
    )
         << endl;

    cout << "This estimate is valid for " << estimate_expiry << " days" << endl;

    return 0;
}
```