Introduction to C++ Programming

with examples and
hands-on exercises

WEBUCATOR

Version: 1.0.2

The Authors

Jared Dunn

In addition to supporting students in self-paced, online courses and writing and tech editing courseware, Jared is a Full Stack Developer at Webucator. He builds and maintains many of Webucator's internal and external applications. He has a BA in Computer Science and a BA in International and Public Affairs from Brown University.

Stephen Withrow

Stephen has over 30 years of experience in training, development, and consulting in a variety of technology areas including Python, Java, C, C++, XML, JavaScript, Tomcat, JBoss, Oracle, and DB2. His background includes design and implementation of business solutions on client/server, Web, and enterprise platforms. Stephen has a degree in Computer Science and Physics from Florida State University.

Jared Dunn (Editor)

In addition to supporting students in self-paced, online courses and writing and tech editing courseware, Jared is a Full Stack Developer at Webucator. He builds and maintains many of Webucator's internal and external applications. He has a BA in Computer Science and a BA in International and Public Affairs from Brown University.

Amna Salman (Editor)

Class Files

Download the class files used in this manual at https://static.webucator.com/media/public/materials/classfiles/CPP120-1.0.2-introduction-to-c-programming.zip.

Table of Contents

LESSON 1. Getting Started with Modern C++....1
History and Evolution of C++....1
Procedural and Object-Oriented Features....3
Exercise 1: First Program Walkthrough....8
Program Structure Basics....10
LESSON 2. Core Syntax and Control Flow....13
Variables, Constants, and Type Deduction....14
Operators and Expressions....28
Basic Input and Output....36
Modern Output: std::format....42
Conditionals: if and switch....49
Loops: for, while, do-while....63
Exercise 2: Control Flow Practice....73
LESSON 3. Functions and Functional Features....79
Defining and Using Functions....80
Overloading and Default Parameters....84
Scope and Lifetime....88
Lambdas and Generics....94
consteval and constexpr....99
Error Handling Techniques....104
Exercise 3: Function and Scope Practice....108
LESSON 4. Arrays and Strings....113
C-style Arrays and Strings Overview....114
Modern Arrays: std::array....123
Strings with std::string and std::string_view....130
String Formatting with std::format....136
Exercise 4: Exploring Arrays and Strings in C++....146
LESSON 5. Memory and Pointers....153
Basics of Pointers and Addresses....154
Pointer Arithmetic....162
Dynamic Memory: new and delete....168
Smart Pointers: unique_ptr, shared_ptr, weak_ptr....172
Pointers vs. References....179
Exercise 5: Explore Pointers and Memory....186

LESSON 6. Structured Programming....189
User-Defined Types: struct....189
typedef and enum....198
Passing Structs to Functions....207
Exercise 6: Practice with User-Defined Types and Enums....216
LESSON 7. Standard Data Structures....225
Vectors and Lists....226
Stacks, Queues, and Deques....234
Maps and Sets....243
Unordered Containers....251
Common Algorithms: sort, find, Ranges....257
Exercise 7: Explore and Implement Standard Data Structures....266
LESSON 8. Object-Oriented Programming....275
OOP Principles: Encapsulation, Inheritance, Polymorphism....276
Class Definitions....278
Member Variables and Functions....282
Constructors and Destructors....287
Access Control....293
Function and Operator Overloading....298
Exercise 8: Designing a Basic OOP System....306
LESSON 9. Inheritance and Polymorphism....317
Base and Derived Classes....318
Inheritance Types....327
Virtual Functions and Dynamic Dispatch....335
Abstract Classes and Interfaces....346
Exercise 9: Inheritance and Polymorphism Practice....356
LESSON 10. File I/O....369
File Streams and Concepts....370
Reading Files: ifstream....372
Writing Files: ofstream....377
File Modes and Error Handling....382
Working with Structured Data Files....390
Exercise 10: Practice File I/O....409

LESSON 1

Getting Started with Modern C++

Topics Covered

☑ How C++ evolved from C.

☑ Basics of procedural programming vs Object-Oriented Programming.

☑ Step-by-step walkthrough of writing, compiling, and running a first program.

☑ Program structure essentials: headers, namespaces, main(), and functions.

Introduction

In this lesson you will be introduced to modern C++. You will learn about the history of the programming language. Basic examples will be presented as well.

EVALUATION COPY: Not to be used in class.

1.1. History and Evolution of C++

❖ 1.1.1. What is C++?

C++ is a language for telling a computer exactly what to do. It helps us write clear, step-by-step instructions so programs run quickly and correctly. People use C++ for games, web browsers, robots, cars, operating systems, and more because it gives speed and precise control.

EVALUATION COPY: Not to be used in class.

❖ 1.1.2. Where C++ fits among other languages

Different languages make different tradeoffs. Languages like Python or JavaScript are easy to start with but can run slower. C++ asks you to be a bit more careful, and in return it can run very fast and give you fine control over memory. Some languages have a garbage collector, which is automatic memory clean-up. C++ usually asks you to manage memory directly. Modern C++ provides safe helpers called smart pointers that make this easier. We will learn these tools in this course.

❖ 1.1.3. Where did C++ come from?

C++ grew out of an older language called C. C is one of the most important languages in computing and influenced many others. In the early 1980s, Bjarne Stroustrup created "C with Classes" to add new ways to organize code. One major idea is object-oriented programming, which we will cover in this course. In 1983 the language was named C++. Since then, new versions have added features that improve safety and clarity while keeping the speed and control that made C++ popular.

❖ 1.1.4. Timeline

C++98

Released in 1998, C++98 was the first official version of the language. It set the foundation with features such as templates, exceptions, and namespaces. These features helped developers organize code better and handle errors more safely.

C++03

Released in 2003, C++03 made small updates and fixes based on feedback from developers using C++98. It focused on improving reliability rather than adding new features.

C++11

C++11 was a major update. It introduced features like the `auto` keyword, lambda expressions, smart pointers, and range-based for loops. These changes made the language easier to use, more powerful, and more efficient for modern programming.

C++14

C++14 built on C++11 by refining what was already there. It made syntax cleaner, fixed bugs, and improved performance. It helped programmers write modern C++ code more smoothly.

C++17

C++17 expanded the standard library and focused on faster, more flexible programs. It added optional types, new parallel algorithms, and a filesystem library, making it easier to write efficient and organized code.

C++20

C++20 brought some of the biggest improvements in the history of the language. It added concepts, ranges, coroutines, and modules. These features make C++ code easier to read, safer to use, and better suited for large projects.

C++23

C++23 continues to build on what came before. It adds more helpful library functions, improves consistency, and gives compilers better tools to understand and optimize programs. Each new version moves C++ closer to being both powerful and beginner-friendly.

What will use in this course?

In this course, we will focus on using **C++20**. It offers a strong balance of speed, safety, and modern design. Learning C++20 will give you a solid foundation for understanding how C++ has evolved and how it is used in real-world projects today.

1.2. Procedural and Object-Oriented Features

Programming languages can be used in different ways to organize and structure code. Two common ways are **procedural programming** and **object-oriented programming (OOP)**. C++ supports both, which makes it a flexible language that fits many kinds of projects.

C++ 20 Used In This Course

We will use C++ 20 in this course. Although C++ 23 is available, support for this standard is not universal yet.

In **procedural programming**, you write programs as a series of steps or procedures that the computer follows in order. Each procedure (called a **function** in C++) performs a specific task. This approach is often used for small programs or when the goal is to describe a clear sequence of actions.

In **object-oriented programming**, code is organized around **objects** instead of steps. Objects are created from **classes**, which describe what data an object holds and what actions it can perform. This approach is helpful for modeling real-world things like a car, a player in a game, or a bank account. It also helps keep large programs organized and easier to maintain.

You can think of it this way: procedural programming focuses on **what to do next**, while object-oriented programming focuses on **who or what is doing the work**. Both styles are important, and as you learn C++, you'll see how they can work together.

Make sure that you're set up!

Before continuing, be sure that you've completed the setup instructions.

❖ 1.2.1. Procedural Programming in C++

Procedural programming in C++ uses functions to divide a task into smaller parts. Each function does one job, such as printing a message or calculating a value. Functions can be placed inside the main program or stored in separate files that work together.

Here is a simple example of a procedural program:

Demo 1.1: GettingStartedModernC/Demos/MyFirstConsoleApp.cpp

```
1.   #include <iostream>
2.
3.   int main() {
4.       std::cout << "This is my first C++ program!\n";
5.       return 0;
6.   }
```

This program displays a message in the console, which is the text window where programs can print information. When you run it, you will see:

```
This is my first C++ program!
```

Running this program on Windows:

1. Open Visual Studio Code.
2. From the **File** Menu select **Open Folder...**.
3. Navigate to `GettingStartedModernC\Demos`. Click **Select Folder**.
4. The folder will appear in the **Explorer** view in the left hand panel. Select `SimpleOOP.cpp` so that the file will open in the editor.
5. Run the file by pressing **Ctrl** + **F5**.
6. The output will display in the terminal at the bottom of the screen.

Running this program on Mac:

1. Follow Steps 1 through 4 above for running on Windows.
2. Right click the `Demos` folder in the **Explorer** panel. Select **Open In Integrated Terminal**.
3. You will now see a terminal view beneath the editor. Copy and paste the following command into the terminal and press **Enter**.

```
clang++ -std=c++20 MyFirstConsoleApp.cpp -o output
```

4. Run the program by typing the following into the terminal and pressing **Enter**:

```
./output
```

❖ 1.2.2. Object-Oriented Programming in C++

Object-oriented programming (OOP) organizes code around **objects**, which represent things tha have data and behavior. Objects are created from classes, which act as blueprints for those objects.

C++ supports four main ideas of OOP: **inheritance**, **polymorphism**, **abstraction**, and **encapsulation**. These ideas make code easier to reuse, extend, and maintain. You don't need to understand these terms yet. We will explain them later in the course.

Here is a simple example of an object-oriented program in C++:

Demo 1.2: GettingStartedModernC/Demos/SimpleOOP.cpp

```
1.   #include <iostream>
2.   #include <format>
3.
4.   class Flower {
5.   private:
6.       std::string typeOfFlower;
7.
8.   public:
9.       // Constructor with type of flower
10.      Flower(const std::string& tFlower)
11.          : typeOfFlower(tFlower) {
12.      }
13.
14.      std::string getTypeOfFlower() {
15.          return typeOfFlower;
16.      }
17.
18.      std::string display() {
19.          return std::format("I am a {}!\n",getTypeOfFlower());
20.      }
21.  };
22.
23.  int main() {
24.      Flower myFlower = Flower("Red Rose");
25.      std::cout << myFlower.display();
26.  }
```


Here is the output:

```
I am a Red Rose!
```

Running this program on Windows:

Follow the same instructions as you did for the `MyFirstConsoleApp.cpp` program, but open `SimpleOOP.cpp` in the editor this time.

Running this program on Mac:

Follow the same instructions as before but use the `SimpleOOP.cpp` in the compile command:

```
clang++ -std=c++20 SimpleOOP.cpp -o output
```


❖ 1.2.3. Summary

Procedural and object-oriented programming are two ways of thinking about how to structure code. Procedural programming focuses on **functions and steps**, while object-oriented programming focuses on **objects and relationships**. Both are powerful, and C++ gives you the tools to use either approach (or both together) as you build larger, more flexible programs. We will explore them both throughout this course.

Exercise 1: First Program Walkthrough

5 to 15 minutes

In this activity, you will write, compile, and run a small C++ program that writes a message to the console.

Open your editor and type the following code:

```
#include <iostream>

int main() {
    std::cout << "This is my first C++ program!\n";
    return 0;
}
```

Save your file in `GettingStartedModernC\Exercises\MyFirstConsoleApp.cpp`.

What each part does:

- `#include <iostream>`: Adds the standard input-output library, which is necessary for writing to the console with `std::cout`.
- `int main()`: Defines the main function where the program execution begins. `int` indicates that the function returns an *integer* data type. We will discuss different data types in the next lesson.
- `std::cout`: An object from C++'s built-in collection of features, called the *standard library*.
    - `std` is the "namespace" of the standard library. Namespaces ensure that if multiple libraries have features with the same name, you can still distinguish between them.
    - `cout` (short for "character output") is an object that can be used to send text to the console.
- `<<` (insertion operator): Sends the text on its right into `std::cout` so it appears on the console.
- `"This is my first C++ program!\n"`: The message to print. `\n` is a newline character that moves the cursor to the next line.
- `;` (semicolon): Ends a statement. C++ requires a semicolon at the end of most statements.
- `return 0;`: Returns `0` from `main`, which indicates the program finished successfully.

Compile and run your program using the steps from the previous activity for your operating system (Windows or Mac).

If everything is correct, the output will be:

```
This is my first C++ program!
```

Solution: GettingStartedModernC/Demos/MyFirstConsoleApp.cpp

```
1.   #include <iostream>
2.
3.   int main() {
4.       std::cout << "This is my first C++ program!\n";
5.       return 0;
6.   }
```

Running this program on Windows:

1. Open Visual Studio Code.
2. From the File Menu select **Open Folder...**
3. Navigate to `GettingStartedModernC\Demos`. Click **Select Folder**.
4. The folder will appear in the Explorer view in the left hand panel. Select `MyFirstConsoleApp.cpp` so that the file will open in the editor.
5. Run the file by pressing **Ctrl** + **F5**.
6. The output will display in the terminal at the bottom of the screen.

Running this program on Mac:

1. Follow Steps 1 through 4 above for running on Windows.
2. Right click the Demos folder in the Explorer panel. Select **Open In Integrated Terminal**.
3. You will now see a terminal view beneath the editor. Copy and paste the following command into the terminal and press **Enter**.

```
clang++ -std=c++20 MyFirstConsoleApp.cpp -o output
```

4. Run the program by typing the following into the terminal and pressing **Enter**:

```
./output
```


1.3. Program Structure Basics

Let's review the basics of structuring your C++ program.

You may refer to the following demo C++ program (shown earlier) throughout this activity.

Demo 1.3: GettingStartedModernC/Demos/MyFirstConsoleApp.cpp

```
1.  #include <iostream>
2.
3.  int main() {
4.      std::cout << "This is my first C++ program!\n";
5.      return 0;
6.  }
```


❖ 1.3.1. main Function

The `main` function is the entry point of any C++ program. When the program is executed, the code inside this function is executed first. The `main` function is required in every C++ program that is run from the command line or terminal view. This type of program is referred to as a **console application**.

The curly braces (`{ }`) define the scope of the function, in other words, the statements that make up the function.

The `return` statement returns a value to the calling function. Because `main` is automatically called by the runtime environment, the return value (which is `0`) in the demo program above will be displayed on the console.

❖ 1.3.2. Headers

Headers are files that contain definitions of functions that can be used in a C++ program. The `#include` directive is used to incorporate header files, like `#include <iostream>` for standard input and output operations.

❖ 1.3.3. Preprocessor Directives

Preprocessor directives are lines included in the code of programs preceded by a hash tag (#). They are not part of the core C++ language but are used to instruct the compiler to process certain tasks before actual compilation begins. Common directives include `#include` and `#define`.

❖ 1.3.4. Namespaces

Namespaces are used to organize code into logical groups and prevent name collisions that can occur especially when your code base includes multiple libraries. The most common namespace is `std`, which includes features of the C++ Standard Library.

Conclusion

In this lesson, "Getting Started," we have explored the foundational concepts and skills needed to embark on your journey. We began by understanding the importance of setting clear goals and how they guide your path. The lesson highlighted essential tools and resources available to support your learning experience. Through practical exercises, you engaged in initial tasks that set the groundwork for more advanced topics. Emphasis was placed on the value of continuous learning and adaptability in a rapidly changing environment. As we conclude, remember that the insights and skills gained here are only the beginning. Use them as stepping stones to propel you forward, encouraging a mindset of curiosity and determination as you navigate your learning journey.

LESSON 2

Core Syntax and Control Flow

Topics Covered

☑ Declaring variables and constants.

☑ Building expressions and using operators.

☑ Performing basic input and output with standard streams.

☑ Formatting text with std::format for modern output.

☑ Making decisions with if/else and switch statements.

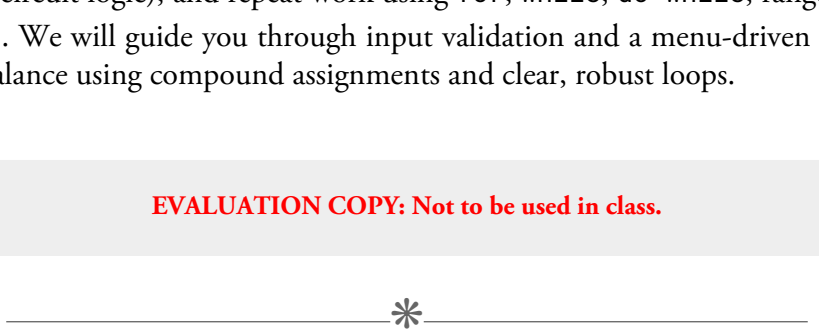
☑ Iterating with for, while, and do-while loops.

Introduction

In this lesson, you will build a solid foundation in C++ core syntax and control flow. You'll declare variables and constants, choose the right data types, and use type deduction with `auto`. You'll compose expressions with arithmetic, comparison, logical, assignment, and ternary operators, watch out for precedence, and concatenate strings confidently.

You will also perform input/output with `std::cout` and `std::cin`, read full lines safely, and format output with `std::format`. Then you'll direct program behavior using `if/else` and `switch` (including enums and short-circuit logic), and repeat work using `for`, `while`, `do-while`, range-based `for`, plus `break/continue`. We will guide you through input validation and a menu-driven mini project that updates a bank balance using compound assignments and clear, robust loops.

EVALUATION COPY: Not to be used in class.



EVALUATION COPY: Not to be used in class.

2.1. Variables, Constants, and Type Deduction

In programming, a **variable** is like a labeled box that stores a piece of information for your program to use. You give the box a name (the variable's name) and decide what kind of thing it will hold (its data type). The computer remembers the value you store there, and you can reuse or change that value as your program runs.

When you create a variable in C++, you are telling the computer three things:

1. **What kind of data** the variable will hold (the data type).
2. **What to call it** (the variable name).
3. **What value to start with** (the initial value, which is optional).

Here's what a simple variable definition looks like:

```
int score = 10; // defines an integer variable named score and gives it a starting value
 of 10
```


You can think of this as saying: "Create a box called `score` that holds whole numbers, and put `10` inside it."

Once defined, you can update the value in that box by using an **assignment statement**:

```
score = 15; // changes the value stored in score from 10 to 15
```

However, you cannot redefine the same variable name in the same block of code: each variable name must be unique within its scope:

```
int score = 10;
int score = 20; // error: score is already defined
```

Later, you'll learn how **constants** work. They're like variables, but their values can't be changed after being set: useful for things like mathematical constants or fixed limits in your program.

Next, let's explore the different **data types** that determine what kind of information a variable can hold.

❖ 2.1.1. Data Types

In C++, every variable has a *data type,* a label that tells the computer what kind of value the variable will store (like whole numbers, numbers with decimals, text, or true/false). Choosing a data type helps the computer decide how much memory to reserve and what operations are allowed. C++ is a **statically typed** (also sometimes called "strongly typed") language, which means once you declare a variable with a type, that type doesn't change. You can update the variable's value later, but it must stay compatible with the original type.

To create a variable, write the type first, then the variable name, and (optionally) give it a starting value:

```
int myInt = 45; // type: int, name: myInt, initial value: 45
```

Below is a list of common C++ data types. For each one, you'll see:

1. a simple definition to build an intuitive understanding,
2. a technical definition with the precise details you'll need as you progress,
3. and a few examples of what using that data type might look like in practice.

Integer (`int`)

Simple Definition:

A whole number (no decimals) like the numbers you use for counting or keeping score: 0, 1, 400, -3, etc.

Technical Definition:

A 32-bit integer on most systems, storing values between **–2,147,483,648** and **2,147,483,647**. You can write integers in **decimal (base 10)**, **binary (base 2)** using `0b`, or **hexadecimal (base 16)** using `0x`.

Examples:

```
int myInt = 15;        // decimal (base 10)
int binary15 = 0b1111; // binary (base 2) => 15 in decimal
int hex15 = 0xF;       // hexadecimal (base 16) => 15 in decimal
```

Decimal, Binary, and Hexadecimal

When we write numbers in everyday life (like **10**, **25**, or **300**),we're using the **decimal system**, which is based on **ten digits (0–9)**. Computers, however, store and work with data in **binary (base 2)**, numbers made up of only 0s and 1s. Because binary numbers can get long and hard to read, programmers sometimes use **hexadecimal (base 16)**, a shorter, more compact way to represent binary values.

Throughout this course, **you will mostly use normal decimal numbers**, but if you would like to understand decimal, binary, and hexadecimal in more depth, take a look at the below explanations.

Decimal (Base 10)

This is the number system you already know. Each place value represents a power of 10.

Powers of 10:

- $10^0 = 1$.
- $10^1 = 10$.
- $10^2 = 100$.

245 means:

$(2 \times 10^2) + (4 \times 10^1) + (5 \times 10^0) = (2 \times 100) + (4 \times 10) + (5 \times 1) = 200 + 4 + 5 =$ **245**

Binary (Base 2)

Binary only uses two digits: **0** and **1**. Each position represents a power of **2**, instead of 10.

Powers of 2:

- $2^0 = 1$
- $2^1 = 2$
- $2^2 = 4$
- $2^3 = 8$

Let's take the example of 0b1111, which is equivalent to 15 in decimal:

$(1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 8 + 4 + 2 + 1 =$ **15**

Hexadecimal (Base 16)

Hexadecimal (or hex for short) uses 16 symbols to represent values:

0-9 represent 0 through 9, and A-F represent 10 through 15.

So in C++, 15 in hex would be written as 0xF because F represents 15.

Powers of 16:

- $16^0 = 1$
- $16^1 = 16$
- $16^2 = 256$

Let's take the example of 0x2A, which is equivalent to 42 in decimal:

$(2 \times 16^1) + (9 \times 16^0) = (2 \times 16) + (9 \times 1) = 32 + 9 =$ **42**

Long Integer (`long long`)

Simple Definition:

A much bigger whole number used when a regular `int` isn't large enough. It is often just called a "long."

Technical Definition:

A 64-bit integer (usually twice as large as an `int`) that can hold very big positive or negative numbers, roughly from **–9,223,372,036,854,775,808** to **9,223,372,036,854,775,807**.

Examples:

```
long long bigCount = 214748364811;
long long veryBig = 9000000000000000000LL; // large number
```

Longs should have the `LL` or `ll` suffix.

Floating-Point Number (`float`)

Simple Definition:

A number that can include a decimal point, like 3.14 or -0.5. Use it for values that don't need to be exact, such as averages or measurements. It is often just called a "float."

Technical Definition:

A number type that can represent very large or very small decimal values using scientific notation. It has about **7 digits of precision** and can store numbers roughly between **−3.4 × 1038** and **+3.4 × 1038**.

Examples:

```
float pi = 3.1415926f; // about 7 digits of precision
float huge = 3.4e38f; // 3.4 × 10^38
float tiny = 1.2e-38F; // very small number
```

Floats should have the `f` or `F` suffix.

Double-Precision Number (`double`)

Simple Definition:

Similar to a `float` but can hold larger and more precise decimal numbers. It is often just called a "double."

Technical Definition:

A number type that provides about **15 to 16 digits of precision** and can store extremely large or small values, roughly between **−1.8 × 10308** and **+1.8 × 10308**.

Examples:

```
double price = 19.99; // default type for decimal numbers
double distance = 3.4e40; // 3.4 × 10^40
double precise = 0.1234567890123456; // more precision than float
```

Decimal numbers are treated as doubles by default so no suffix is specified.

String (`std::string`)

Simple Definition:

A piece of text that can include letters, numbers, spaces, or symbols. Strings are written inside double quotes (`" "`).

Technical Definition:

A text data type provided by the C++ Standard Library (`#include <string>`). It represents a sequence of characters that can grow or shrink in size. Use `std::string` because the type lives inside the `std` (standard) namespace.

Examples:

```
#include <string>

std::string name = "Stephen";
std::string message = "Hello, " + name + "!";
std::string special = "A1! @#?";
```

Character (`char`)

Simple Definition:

A single character (such as a letter, number, or symbol) written inside single quotes (`' '`). Strings are made up of characters.

Technical Definition:

A data type that stores a single character using one byte (8 bits) of memory. Each character is represented by a numeric code, typically following the ASCII standard. For example, the letter `'A'` has a numeric value of 65. You can use characters like `'a'`, `'5'`, `'!'`, or even special escape sequences like `'\n'` for a new line.

Examples:

```
char grade = 'A'; // single character
char digit = '7'; // character '7', not the number 7
char symbol = '!'; // punctuation
char newline = '\n'; // special character: new line
```

Boolean (`bool`)

Simple Definition:

A true-or-false value: like an on/off switch or a yes/no question.

Technical Definition:

A type that can store only two possible values: `true` (represented as 1) or `false` (represented as 0). Any nonzero number automatically becomes `true`.

Examples:

```
bool isReady = true; // true
bool isZero = 0; // false
bool fromNum = 2025; // true (nonzero becomes true)
```

Size Type (`size_t`)

Simple Definition:

`size_t` is a type of number used to count things, like how many items are in an array. It can only be 0 or positive. It is commonly used for array indices, loop counters, and any situation where a value cannot be negative.

Technical Definition:

`size_t` is defined by the C++ standard as an unsigned integer type that is guaranteed to be able to represent the size of any object in bytes. On most computers, it is either 32-bit or 64-bit.

Examples:

```
size_t count = 5;    // count of items
for (size_t i = 0; i < count; i++) {
    std::cout << "Item " << i << "\n";
}
size_t arraySize = sizeof(int) * 5; // size of 5 integers in bytes
```

Converting Between Data Types

Sometimes in programming, you need to change a value from one type to another. This could be converting a number to a different kind of number, or turning a string that contains a number into an actual numeric value. This section shows simple ways to do these conversions safely.

`static_cast`

- **Simple Definition:**

- A tool in C++ that changes a value from one numeric type to another, for example turning an `int` into a `double`.

- **Technical Definition:**
  - An explicit type conversion operator in C++ that safely converts between compatible data types, including numeric types, pointers, and references.

- **Example:**

```
int i = 10;
double d = static_cast<double>(i); // int to double
```

`std::stoi`

- **Simple Definition:**
  - Converts a string to an integer.

- **Technical Definition:**
  - Reads a `std::string` and returns its value as `int`. Throws an exception if conversion fails.

- **Example:**

```
std::string str = "123";
int num = std::stoi(str); // string to int
```

`std::stod`

- **Simple definition:**
  - Converts a string to a double.

- **Technical definition:**
  - Reads a `std::string` and returns its value as `double`. Throws an exception if conversion fails.

- **Example:**

```
std::string str = "3.14159";
double d = std::stod(str); // string to double
```

`std::stol`

- **Simple definition:**
  - Converts a string to a long integer.
- **Technical definition:**
  - Reads a `std::string` and returns its value as `long`. Throws an exception if conversion fails.
- **Example:**

```
std::string str = "123456";
long num = std::stol(str); // string to long
```


`std::stoll`

- **Simple definition:**
  - Converts a string to a long long integer.
- **Technical definition:**
  - Reads a `std::string` and returns its value as `long long`. Useful for very large integers.
- **Example:**

```
std::string str = "1234567890123";
long long num = std::stoll(str); // string to long long
```

`std::stof`

- **Simple definition:**

- Converts a string to a float.

- **Technical definition:**
  - Reads a `std::string` and returns its value as `float`. Throws an exception if conversion fails.

- **Example:**

```
std::string str = "3.14";
float f = std::stof(str); // string to float
```

`std::to_string`

- **Simple definition:**
  - Converts a number to a string.

- **Technical definition:**
  - Takes numeric types like `int`, `long`, `float`, `double` and returns a `std::string` representing the value.

- **Example:**

```
int i = 42;
std::string str = std::to_string(i); // int to string
```

❖ 2.1.2. Type Deduction (with `auto`)

You can use the auto keyword to let the compiler figure out the variable type rather than assigning it yourself. This is called **type deduction**.

Why use it?

- Reduces repetition: you don't have to type long or complex types.
- Keeps code flexible if the initializer's type changes later.

How it works:

- You **must** provide an initializer; otherwise the compiler has nothing to deduce from.

- The deduced type is the type of the initializer.

Common examples

```
auto count = 42;   // int
auto big = 42LL;   // long long (because of the LL suffix)
auto rate = 3.5;   // double (decimal numbers are doubles by default)
auto rateF = 3.5f; // float (deduced with the f suffix)
```

❖ 2.1.3. Constants (using `const`)

Constants are variables that cannot be changed after initialization. They are effectively **read-only**. Using constants provides safety by preventing accidental modification of critical values. It also makes it really clear to the reader: "this value won't change."

The standard naming convention for constants is `ALL_CAPS_WITH_UNDERSCORES`:

Examples

```
const float TAX_RATE = 0.07f;
const int MAX_RETRIES = 3;
const double PI = 3.14159;
```

Ready to see these concepts in action? Test out the demo below to explore examples and try them out yourself.

Demo 2.1: CoreSyntaxControlFlow/Demos/VariablesConstantsTypeDeduction.cpp

```
1.   #include <iostream>
2.   #include <string>
3.
4.   int main() {
5.       // --- Integer (int) ---
6.       int myInt = 45;                 // decimal (base 10)
7.       int binary15 = 0b1111;          // binary (base 2) => 15 in decimal
8.       int hex15 = 0xF;                // hexadecimal (base 16) => 15 in decimal
9.
10.      // --- Long Integer (long long) ---
11.      long long myLongInt = 2147483648LL; // value too large for int, use LL for
             long long
12.
13.      // --- Floating-Point Number (float) ---
14.      float myFloat = 3.4e37F;        // 3.4 Ã— 10^37 (use f or F suffix for float)
15.
16.      // --- Double-Precision Number (double) ---
17.      double myDouble = 3.4e40;       // larger range and precision than float
18.      double price = 19.99;           // default for decimals is double
19.
20.      // --- Character (char) ---
21.      char grade = 'A';               // single character
22.      char symbol = '!';              // punctuation character
23.      char newline = '\n';            // special escape sequence (new line)
24.
25.      // --- String (std::string) ---
26.      std::string myName = "Stephen";
27.      std::string greeting = "Hello, " + myName + "!";
28.
29.      // --- Boolean (bool) ---
30.      bool isReady = true;            // true (1)
31.      bool isZero = 0;                // false (0)
32.      bool fromNum = 2025;            // true (nonzero converts to true)
33.
34.      // --- Constants (const) ---
35.      const double PI = 3.14159;      // value cannot change after initialization
36.      const int MAX_RETRIES = 3;
37.      const float TAX_RATE = 0.07F;
38.
39.      // --- Type Deduction (auto) ---
40.      auto myHomeTown = "Warner Robins"; // deduced as const char*
41.      auto year = 2025;                  // deduced as int
42.      auto bigNumber = 9'000'000'000LL;  // deduced as long long
```

Demo 2.1: CoreSyntaxControlFlow/Demos/VariablesConstantsTypeDeduction.cpp

```
43.       auto discountRate = 0.15;          // deduced as double
44.
45.       // --- Reassignment Example ---
46.       std::cout << "Original myInt value: " << myInt << "\n";
47.       myInt = 100;                       // reassigning a new value to myInt
48.       std::cout << "Reassigned myInt value: " << myInt << "\n\n";
49.
50.       // --- Constants Cannot Be Reassigned ---
51.       // The following examples would cause compile-time errors if uncommented:
52.       /*
53.       PI = 3.14;          // error: cannot modify a const variable
54.       TAX_RATE = 0.05F;   // error: cannot change value of constant
55.       */
56.
57.       // --- Output all values ---
58.       std::cout << "Here are the values of the variables defined in this pro  ↵↵
               gram:\n";
59.       std::cout << "int myInt: " << myInt << "\n";
60.       std::cout << "binary15 (0b1111): " << binary15 << "\n";
61.       std::cout << "hex15 (0xF): " << hex15 << "\n";
62.       std::cout << "long long myLongInt: " << myLongInt << "\n";
63.       std::cout << "myFloat: " << myFloat << "\n";
64.       std::cout << "myDouble: " << myDouble << "\n";
65.       std::cout << "price: " << price << "\n";
66.       std::cout << "char grade: " << grade << "\n";
67.       std::cout << "char symbol: " << symbol << "\n";
68.       std::cout << "myName: " << myName << "\n";
69.       std::cout << "greeting: " << greeting << "\n";
70.       std::cout << "bool isReady: " << isReady << "\n";
71.       std::cout << "bool isZero: " << isZero << "\n";
72.       std::cout << "bool fromNum: " << fromNum << "\n";
73.       std::cout << "PI (constant): " << PI << "\n";
74.       std::cout << "MAX_RETRIES (constant): " << MAX_RETRIES << "\n";
75.       std::cout << "TAX_RATE (constant): " << TAX_RATE << "\n";
76.       std::cout << "auto myHomeTown: " << myHomeTown << "\n";
77.       std::cout << "auto year: " << year << "\n";
78.       std::cout << "auto bigNumber: " << bigNumber << "\n";
79.       std::cout << "auto discountRate: " << discountRate << "\n";
80.
81.       return 0;
82.   }
```


Here's a refresher on how to run the file:

1. Open Visual Studio Code.
2. From the **File** menu, select **Open Folder**. Navigate to `CoreSyntaxAndControlFlow\Demos`.
3. Click **OperatorsAndExpressions.cpp** to open the file in the editor.
4. Right click the `Demos` folder in the **Explorer** panel. Select **Open In Integrated Terminal**.
5. Compile and run the program:
    A. Windows: press **Ctrl** + **F5** to run the program.
    B. Mac: run the following commands in the terminal view:

```
clang++ -std=c++20 OperatorsAndExpressions.cpp -o output
./output
```


2.2. Operators and Expressions

In programming, an **expression** is a combination of values, variables, and **operators** that the computer evaluates to produce a result. You've already seen values and variables. Operators are the symbols (like `+`, `-`, or `==`) that tell the computer what kind of work to do with those values.

Think of operators as tools, and expressions as little "calculations" that use those tools. In this reading, you'll learn the most common operator types in C++ and how to use them safely and clearly.

❖ 2.2.1. Arithmetic Operators

These work like the math you already know and produce numeric results.

- `+` add
- `-` subtract
- `*` multiply
- `/` divide
- `%` modulus - calculates the remainder of a division, only for integers

```
int a = 7, b = 3;
int sum = a + b;        // 10 (sum)
int diff = a - b;       // 4 (difference)
int prod = a * b;       // 21 (product)
int quotInt = a / b;    // 2  (integer division: decimals are discarded)
int rem = a % b;        // 1  (remainder)
```

Integer vs. Floating-Point Division

When both sides of `/` are integers, the result is an integer, and any fractional part is **discarded**. To get a decimal result, make at least one side a float or a double.

❖ 2.2.2. Increment and Decrement

These change a value by 1.

- `++x` pre-increment: add 1, then use the value
- `x++` post-increment: use the value, then add 1
- `--x` and `x--` do the same but subtract 1


```
int x = 5;
int a1 = ++x; // x becomes 6, then a1 gets 6
int a2 = x++; // a2 gets 6, then x becomes 7
```

❖ 2.2.3. Assignment and Compound Assignment

Use `=` to store a value in a variable. Compound assignments do an operation and assignment in one step.

- `=` assign
- `+=`, `-=`, `*=`, `/=`, `%=` do-and-assign

```
int total = 10;
total += 5;  // total = total + 5  => 15
total *= 2;  // total = total * 2  => 30
```

❖ 2.2.4. Comparison Operators

These compare two values and produce a `bool` result: `true` or `false`.

- `==` equal to
- `!=` not equal to
- `<` less than, `<=` less than or equal to
- `>` greater than, `>=` greater than or equal to

```
int p = 5, q = 8;
bool isLess = p < q;   // true (1)
bool isEqual = p == q; // false (0)
```

❖ 2.2.5. Logical Operators

Combine true/false values to make decisions.

- `&&` logical AND: true if both sides are true
- `||` logical OR: true if at least one side is true
- `!` logical NOT: flips true/false


```
bool hasTicket = true;
bool isGuest = false;
bool canEnter = hasTicket || isGuest; // true
bool needsCheck = hasTicket && !isGuest; // true
```

❖ 2.2.6. The Conditional (Ternary) Operator

Selects one of two expressions based on a Boolean condition and yields a value.

```
int a = 5, b = 9;
int max = (a > b) ? a : b; // 9 (b) - because a (5) is not greater than b (9)
```

Syntax: `condition ? expr_if_true : expr_if_false`. If `condition` is true, the result is `expr_if_true`; otherwise it is `expr_if_false`. Only the selected expression is evaluated.

```
int x = 4;
std::string isEven = (x % 2 == 0) ? "even" : "odd"; // "even" - 4 is even
```

❖ 2.2.7. Operator Precedence and Parentheses

Some operators are evaluated before others. For example, multiplication happens before addition. Parentheses let you control the order and make code easier to read.

```
int i = 2 + 3 * 4;    // 14 (3*4 happens first)
int j = 2 + (3 * 4)   // 14 preferred...leaves nothing to chance!
int k = (2 + 3) * 4;  // 20 (5 * 4)
```

Always use parentheses when you have more than one type of arithmetic operator as j and k demonstrates above. DO not rely on the default order of evaluation. Use of parentheses enhances the accuracy and readability of your code.

The + operator can also be used to join strings. This is called **concatenation**.


```
#include <string>
std::string first = "Ada";
std::string last = "Lovelace";
std::string full = first + " " + last; // "Ada Lovelace"
```

Numbers vs. Text

`'7'` is the character seven, not the number 7. Text goes in quotes. If you mix numbers and text with +, at least one side must be a `std::string` so the result is string concatenation.

❖ 2.2.8. Common Pitfalls to Avoid

- Division by zero. Always check the denominator first.
- Relying on operator precedence. Add parentheses to show intent.
- Using ++ or -- multiple times in one expression. Keep it simple.

Explore the demo below and try changing values to watch how outputs change.

Demo 2.2: CoreSyntaxControlFlow/Demos/OperatorsAndExpressions.cpp

```cpp
#include <iostream>
#include <string>

int main()
{
  std::cout << "=== Operators and Expressions Demo ===\n\n";

  // --------------------------------
  // Arithmetic Operators
  // --------------------------------
  int a = 7, b = 3;
  int sum = a + b;
  int diff = a - b;
  int prod = a * b;
  int quotInt = a / b; // integer division: decimals discarded
  int rem = a % b;     // remainder

  std::cout << "-- Arithmetic Operators --\n";
  std::cout << "a = " << a << ", b = " << b << "\n";
  std::cout << "a + b = " << sum << "\n";
  std::cout << "a - b = " << diff << "\n";
  std::cout << "a * b = " << prod << "\n";
  std::cout << "a / b (int division) = " << quotInt << "\n";
  std::cout << "a % b = " << rem << "\n";

  // --------------------------------
  // Increment and Decrement
  // --------------------------------
  int x = 5;
  int a1 = ++x; // pre-increment: x becomes 6, then a1 gets 6
  int a2 = x++; // post-increment: a2 gets 6, then x becomes 7

  std::cout << "-- Increment and Decrement --\n";
 std::cout << "After pre-increment (++x): a1 = " << a1 << ", x = " << x << "\n";
  std::cout << "After post-increment (x++): a2 = " << a2 << ", x = " << x <<
        "\n\n";

  // --------------------------------
  // Assignment and Compound Assignment
  // --------------------------------
  int total = 10;
  total += 5; // same as total = total + 5
  total *= 2; // same as total = total * 2

```

Demo 2.2: CoreSyntaxControlFlow/Demos/OperatorsAndExpressions.cpp

```
44.     std::cout << "-- Assignment and Compound Assignment --\n";
45.     std::cout << "total = 10; total += 5; total *= 2;\n";
46.     std::cout << "Resulting total = " << total << "\n\n";
47.
48.     // -------------------------------
49.     // Comparison Operators
50.     // -------------------------------
51.     int p = 5, q = 8;
52.     bool isLess = p < q;
53.     bool isEqual = p == q;
54.
55.     std::cout << "-- Comparison Operators --\n";
56.     std::cout << "p = " << p << ", q = " << q << "\n";
57.     std::cout << "p < q: " << isLess << "\n";
58.     std::cout << "p == q: " << isEqual << "\n";
59.     std::cout << "p != q: " << (p != q) << "\n";
60.     std::cout << "p >= q: " << (p >= q) << "\n\n";
61.
62.     // -------------------------------
63.     // Logical Operators
64.     // -------------------------------
65.     bool hasTicket = true;
66.     bool isGuest = false;
67.     bool canEnter = hasTicket || isGuest;
68.     bool needsCheck = hasTicket && !isGuest;
69.
70.     std::cout << "-- Logical Operators --\n";
71.     std::cout << "hasTicket = " << hasTicket << ", isGuest = " << isGuest << "\n";
72.     std::cout << "hasTicket || isGuest = " << canEnter << "\n";
73.     std::cout << "hasTicket && !isGuest = " << needsCheck << "\n\n";
74.
75.     // -------------------------------
76.     // Conditional (Ternary) Operator
77.     // -------------------------------
78.     int num1 = 5, num2 = 9;
79.     int max = (num1 > num2) ? num1 : num2;
80.
81.     int evenTest = 4;
82.     std::string evenOrOdd = (evenTest % 2 == 0) ? "even" : "odd";
83.
84.     std::cout << "-- Conditional (Ternary) Operator --\n";
85.     std::cout << "Between " << num1 << " and " << num2 << ", max = " << max <<
            "\n";
86.     std::cout << evenTest << " is " << evenOrOdd << "\n\n";
```

Demo 2.2: CoreSyntaxControlFlow/Demos/OperatorsAndExpressions.cpp

```
87.
88.     // -------------------------------
89.     // Operator Precedence and Parentheses
90.     // -------------------------------
91.     int n = 2 + 3 * 4;   // multiplication happens first
92.     int m = (2 + 3) * 4; // parentheses change order
93.
94.     std::cout << "-- Operator Precedence and Parentheses --\n";
95.     std::cout << "2 + 3 * 4 = " << n << "\n";
96.     std::cout << "(2 + 3) * 4 = " << m << "\n\n";
97.
98.     // -------------------------------
99.     // String Concatenation
100.    // -------------------------------
101.    std::string first = "Ada";
102.    std::string last = "Lovelace";
103.    std::string full = first + " " + last;
104.
105.    std::cout << "-- Strings and the + Operator --\n";
106.    std::cout << "first: " << first << ", last: " << last << "\n";
107.    std::cout << "full = first + \" \" + last => " << full << "\n\n";
108.
109.    // -------------------------------
110.    // Common Pitfalls
111.    // -------------------------------
112.    std::cout << "-- Common Pitfalls (demonstrations) --\n";
113.    int zero = 0;
114.
115.    // Uncommenting this line would cause a runtime error (division by zero):
116.    // int badDiv = a / zero; // error: division by zero
117.
118.    std::cout << "Always check the denominator before dividing.\n";
119.    std::cout << "Use parentheses to make your intent clear.\n";
120.    std::cout << "Avoid using ++ or -- multiple times in one expression.\n";
121.
122.    std::cout << "\n=== End of Demo ===\n";
123.
124.    return 0;
125. }
```

2.3. Basic Input and Output

Programs are most useful when they can communicate with people. In C++, you'll use the standard input/output library to **display messages** on the screen and **read values** that a user types. This reading introduces the most common tools: `std::cout` for output and `std::cin` for input.

❖ 2.3.1. Getting Started: Headers and Names

To use input and output, include the appropriate headers and use names from the `std` namespace.

```
#include <iostream>  // std::cout, std::cin, std::endl
#include <string>    // std::string (for text input/output)
```

Why std::?

The iostream and string types live inside the C++ Standard Library's `std` namespace. You'll refer to them with the `std::` prefix, like `std::cout` or `std::string`.

❖ 2.3.2. Printing Output with std::cout

You've already seen `std::cout` (pronounced "see-out") in earlier demos to print text and values to the console. This is the standard **output stream** (from `<iostream>`). You have also seen how you can use the **insertion operator** (`<<`) to send data to this output stream.

```
#include <iostream>

int main() {
    std::cout << "Hello, world!\n";  // print a message followed by a newline
    return 0;
}
```

You can chain multiple pieces together with `<<`:

```
int score = 42;
std::cout << "Score: " << score << "\n";
```

To start a new line, use `"\n"` or `std::endl`:

```
std::cout << "Line 1\n";
std::cout << "Line 2" << std::endl; // also moves to a new line
```

In general, prefer `"\n"` because it is shorter to write.

❖ 2.3.3. Reading Input with std::cin

Use `std::cin` to read from the keyboard. The **extraction operator** (`>>`) pulls data from the input stream into your variables.

```
#include <iostream>

int main() {
    int age = 0;

    std::cout << "Enter your age: "; // make sure prompt appears
    std::cin >> age;                // user types a number and presses Enter

    std::cout << "You entered: " << age << "\n";
    return 0;
}
```


You can read multiple values in one line:

```
int x = 0, y = 0;
std::cout << "Enter two integers: ";
std::cin >> x >> y;
std::cout << "You entered x=" << x << " and y=" << y << "\n";
```

Input Types: int, double, char, and std::string

`std::cin` will try to convert what the user types into the variable's type.

```
#include <iostream>
#include <string>

int main() {
    int count = 0;
    double price = 0.0;
    char initial = '\0';       // one character
    std::string word;          // single word (no spaces)

    std::cout << "Enter count (int): ";
    std::cin >> count;

    std::cout << "Enter price (double): ";
    std::cin >> price;

    std::cout << "Enter your initial (char): ";
    std::cin >> initial;

    std::cout << "Enter a single word: ";
    std::cin >> word;          // stops at whitespace

    std::cout << "\nSummary:\n";
    std::cout << "count = " << count << "\n";
    std::cout << "price = " << price << "\n";
    std::cout << "initial = " << initial << "\n";
    std::cout << "word = " << word << "\n";
    return 0;
}
```


Reading Full Lines of Text

`std::cin >> word` stops at spaces. To read a whole line (including spaces), use `std::getline`.

```cpp
#include <iostream>
#include <string>

int main() {
    std::string fullName;

    std::cout << "Enter your full name: ";
    std::getline(std::cin, fullName);           // reads the entire line

    std::cout << "Hello, " << fullName << "!\n";
    return 0;
}
```

Mixing >> and std::getline

If you use `>>` before `std::getline`, a leftover newline may cause `getline` to read an empty line. Fix this by consuming leading whitespace with `std::ws`:


```cpp
int year = 0;
std::string fullName;

std::cout << "Enter graduation year: ";
std::cin >> year;

std::cout << "Enter your full name: ";
std::getline(std::cin >> std::ws, fullName); // eat leftover whitespace

std::cout << "Name: " << fullName << ", Year: " << year << "\n";
```

❖ 2.3.4. Printing Booleans

By default, booleans print as `1` (true) or `0` (false). You can print `true`/`false` with `std::boolalpha`.

```cpp
#include <iostream>

int main() {
    bool isReady = true;
    double total = 19.98765;

    std::cout << "Default bool: " << isReady << "\n"; // 1
    std::cout << std::boolalpha << "Text bool: " << isReady << "\n"; // true
    return 0;
}
```

❖ 2.3.5. Simple Echo Program (Putting It Together)

This small program prompts the user, reads values of different types, and prints a friendly summary.

Demo 2.3: CoreSyntaxControlFlow/Demos/BasicInputOutput.cpp

```
1.   #include <iostream>
2.   #include <iomanip>
3.   #include <string>
4.
5.   int main() {
6.     std::string name;
7.     int items = 0;
8.     double price = 0.0;
9.
10.    std::cout << "Name: ";
11.    std::getline(std::cin, name);
12.
13.    std::cout << "Number of items: ";
14.    std::cin >> items;
15.
16.    std::cout << "Price per item: ";
17.    std::cin >> price;
18.
19.    double cost = items * price;
20.
21.    std::cout << "\n--- Receipt ---\n";
22.    std::cout << "Customer: " << name << "\n";
23.    std::cout << "Items: " << items << "\n";
24.    std::cout << std::fixed << std::setprecision(2);
25.    std::cout << "Total: $" << cost << "\n";
26.
27.    return 0;
28.  }
```


❖ 2.3.6. Common Pitfalls to Avoid

- Forgetting headers: add `#include <iostream>` for `std::cout`/`std::cin` and `#include <string>` for `std::string`.
- Missing `std::` prefix: use `std::cout`, `std::cin`, `std::string`, `std::getline`.
- Using `std::cin >> word` when you meant to read a full sentence. Use `std::getline` for whole lines.

- Mixing `>>` and `std::getline` without handling leftover newlines. Fix with `std::getline(std::cin >> std::ws, line)`.

With these tools, you can start writing programs that ask questions, read answers, and show results.

2.4. Modern Output: std::format

C++20 introduced output formatting using `std::format`. This function offers the following features:

- Interlacing {} in text strings for variable substitution
- Uses **format specifiers** inside the {} for justification, fill, and the number of decimal places for floating point numbers
- Similar to Python's format function and f-string literals

`std::format` is the modern form of the C `printf` function. Advantages of `std::format` as compared to `printf`:

- Arguments are matched to the format specifiers for type safety
- Better compile time error reporting
- Supports `std::string` and `std::string_view`

To use `std::format` include the header `<format>`:

```
#include <format>
```

❖ 2.4.1. `std::format`

The following example demonstrates how to format a line of output before printing it to the console:

```
std::string outputMsg = std::format("I live in {}, {}.\n", myHomeTown, myHomeState);
std::cout << outputMsg;
```

Note that the curly braces (`{}`) serve as a placeholder for a variable. Each placeholder is replaced with the value of the respective variable listed after the string literal.

Here is the output:

```
I live in Warner Robins, Georgia.
```

❖ 2.4.2. Formatting Numbers with `std::format`

Numbers can also be formatted using `std::format`. The curly braces permit you to code **format specifiers**.

Let's take a look at using format specifiers for numbers. We will start by formatting a `float` number:

```
float myFloat = 1234.56789F;  // use f or F when defining a float value
...
std::cout << "My float without formatting: " << myFloat << "\n";
std::cout << std::format("My float  variable with \".2f\" formatting: {0:.2f}\n",
myFloat);
std::cout << std::format("My float variable with \".4f\" formatting: {:.4f}\n", myFloat);
std::cout << std::format("My float variable with \".2e\" formatting: {:.2e}\n", myFloat);
std::cout << std::format("My float variable with \".4e\" formatting: {:.4e}\n", myFloat);
```


Here is the output:

```
My float without formatting: 1234.57
My float  variable with ".2f" formatting: 1234.57
My float variable with ".4f" formatting: 1234.5679
My float variable with ".2e" formatting: 1.23e+03
My float variable with ".4e" formatting: 1.2346e+03
```

Note the following:

1. `{0:.2f}` indicates that the formatting is to be applied to the first variable which is **index** `0` and that **2** digits should be displayed to the right of the decimal point. The index is optional in this case because the variable (`myFloat`) is the first-and only-variable passed to the function.
    - As you saw in the string example presented earlier, you can have more than one format specifier in the string passed to `std::format`. By default, the left to right

sequence of the format specifiers will be applied to the variables passed to the function in that sequence. Therefore, you typically will not have to code the index in the format specifier if your variables are listed in the same sequence as the specifiers that target the variables.

- The number is **rounded** to the nearest hundredths place, therefore `1234.56789` is displayed as `1234.57`.

2. `{:.4f}` specifies that **4** digits should be displayed to the right of the decimal point. The index has been omitted, which is standard operating procedure (refer to the previous note above).
3. `{:.2e}` indicates that the number is to displayed in **scientific notation**, i.e., as a coefficient times a power of 10. The coefficient will be rounded to the nearest hundredths. Accordingly, `1234.56789` becomes `1.23e+03` (1.23 times 10 raised to the positive third power).

The following examples show that `double` numbers can be formatted in the same way as float variables:

```
double myDouble = 123456789.56789;
...
std::cout << "My double without formatting: " << myDouble << "\n";
std::cout << std::format("My double  variable with \".2f\" formatting: {0:.2f}\n", myDou  ↵↵
ble);
std::cout << std::format("My double variable with \".4f\" formatting: {:.4f}\n", myDou  ↵↵
ble);
std::cout << std::format("My double variable with \".2e\" formatting: {:.2e}\n", myDou  ↵↵
ble);
std::cout << std::format("My double variable with \".4e\" formatting: {:.4e}\n", myDou  ↵↵
ble);
```


The output is shown below:

```
My double without formatting: 1.23457e+08
My double  variable with ".2f" formatting: 123456789.57
My double variable with ".4f" formatting: 123456789.5679
My double variable with ".2e" formatting: 1.23e+08
My double variable with ".4e" formatting: 1.2346e+08
```

Binary numbers can also be formatted using `std::format`. Binary numbers consist of digits `0` and `1`:

```
int myBinary = 0B10001111;
...
std::cout << "My binary  without formatting: " << myBinary << "\n";
std::cout << std::format("My binary  variable with \"b\" formatting: {0:b}\n", myBinary);
std::cout << std::format("My binary  variable with \".#b\" formatting: {:#b}\n", myBina ↵↵
ry);
std::cout << std::format("My binary  variable with \"#x\" formatting: {:#x}\n", myBinary);
std::cout << std::format("My binary  variable with \"d\" formatting: {}\n", myBinary);
```

Here is the output:

```
My binary  without formatting: 143
My binary  variable with "b" formatting: 10001111
My binary  variable with ".#b" formatting: 0b10001111
My binary  variable with "#x" formatting: 0x8f
My binary  variable with "d" formatting: 143
```

Note the following:

1. The format specifier is `b`.
2. Use `#b` to display the **identifier**, i.e., `0b` prepended to the number.
3. Use `#x` to display the binary number in **hexadecimal** (base 16) with the identifier.
4. Use `d` to display the binary number in **decimal** (base 10).

`std::format` can be applied to hexadecimal numbers. Hexadecimal digits are `0,1,2...9` and `a` through `f`:

```
int myHexadecimal = 0X8F;
...
std::cout << "My hexadecimal  without formatting: " << myHexadecimal << "\n";
std::cout << std::format("My hexadecimal  variable with \"x\" formatting: {:x}\n", myHex ↵↵
adecimal);
std::cout << std::format("My hexadecimal  variable with \"#x\" formatting: {:#x}\n",
myHexadecimal);
std::cout << std::format("My hexadecimal  variable with \"#b\" formatting: {:b}\n", my ↵↵
Hexadecimal);
std::cout << std::format("My hexadecimal  variable with \"d\" formatting: {}\n", myHex ↵↵
adecimal);
```

The output is shown below:

```
My hexadecimal  without formatting: 143
My hexadecimal  variable with "x" formatting: 8f
My hexadecimal  variable with "#x" formatting: 0x8f
My hexadecimal  variable with "#b" formatting: 0b10001111
My hexadecimal  variable with "d" formatting: 143
```


Note the following:

1. Use `x` as the format specifier to display the hexadecimal digits of the variable.
2. `#` will display the identifier, e.g., `0x`, prepended to the value of the variable.
3. Use `b` to display the value in binary.
4. `d` can be used to display the value of the variable in decimal.

To experiment with the `std::format` function, run the demo program below.

Demo 2.4: CoreSyntaxControlFlow/Demos/Formatting.cpp

```
1.   #include <iostream>
2.   #include <format>
3.
4.   // Define variables
5.   std::string myName = "Stephen";
6.   // use type deduction now for string variables:
7.   auto myHomeTown = "Warner Robins";
8.   auto myHomeState = "Georgia";
9.
10.  float myFloat = 1234.56789F;  // use f or F when defining a float value
11.  double myDouble = 123456789.56789;
12.
13.  int myBinary = 0B10001111;
14.  int binaryOne = 0B00000001;
15.
16.  int myHexadecimal = 0X8F;
17.  int myHexOne = 0X1;
18.
19.
20.  int main() {
21.
22.      // formatting string variables
23.      std::string outputMsg = std::format("I live in {}, {}.\n", myHomeTown, my  ↵↵
             HomeState);
24.      std::cout << outputMsg;
25.
26.      // formatting numbers
27.      // float:
28.      std::cout << "My float without formatting: " << myFloat << "\n";
29.      std::cout << std::format("My float  variable with \".2f\" formatting:
             {0:.2f}\n", myFloat);
30.      std::cout << std::format("My float variable with \".4f\" formatting:
             {:.4f}\n", myFloat);
31.      std::cout << std::format("My float variable with \".2e\" formatting:
             {:.2e}\n", myFloat);
32.      std::cout << std::format("My float variable with \".4e\" formatting:
             {:.4e}\n", myFloat);
33.      // double:
34.      std::cout << "My double without formatting: " << myDouble << "\n";
35.      std::cout << std::format("My double  variable with \".2f\" formatting:
             {0:.2f}\n", myDouble);
36.      std::cout << std::format("My double variable with \".4f\" formatting:
             {:.4f}\n", myDouble);
37.      std::cout << std::format("My double variable with \".2e\" formatting:
             {:.2e}\n", myDouble);
```

Demo 2.4: CoreSyntaxControlFlow/Demos/Formatting.cpp

```
38.       std::cout << std::format("My double variable with \".4e\" formatting:
              {:.4e}\n", myDouble);
39.
40.       std::cout << "My binary  without formatting: " << myBinary << "\n";
41.      std::cout << std::format("My binary  variable with \"b\" formatting: {0:b}\n",
              myBinary);
42.       std::cout << std::format("My binary  variable with \"#b\" formatting:
              {:#b}\n", myBinary);
43.       std::cout << std::format("My binary  variable with \"#x\" formatting:
              {:#x}\n", myBinary);
44.       std::cout << std::format("My binary  variable with \"d\" formatting: {}\n",
              myBinary);
45.
46.      std::cout << "My hexadecimal  without formatting: " << myHexadecimal << "\n";
47.       std::cout << std::format("My hexadecimal  variable with \"x\" formatting:
              {:x}\n", myHexadecimal);
48.       std::cout << std::format("My hexadecimal  variable with \"#x\" formatting:
              {:#x}\n", myHexadecimal);
49.       std::cout << std::format("My hexadecimal  variable with \"#b\" formatting:
              {:#b}\n", myHexadecimal);
50.       std::cout << std::format("My hexadecimal  variable with \"d\" formatting:
              {}\n", myHexadecimal);
51.   }
```


2.5. Conditionals: if and switch

Programs make decisions. Conditionals let your code choose different paths based on true/false tests. In C++, the main tools are the `if` statement and the `switch` statement. You'll use them to respond to input, validate data, and control program flow.

❖ 2.5.1. Booleans and Conditions (Quick Refresher)

A `bool` is either `true` or `false`. Comparison operators like `==`, `<`, and `>=` produce Booleans. Logical operators like `&&`, `||`, and `!` combine or flip them.

```
int age = 20;
bool isAdult = age >= 18;           // true
bool hasID = true;
bool canEnter = isAdult && hasID;   // true AND true => true
```

Short-Circuiting

`&&` and `||` evaluate from left to right and stop early when the result is already known. For example, in `(x != 0 && 10 / x > 2)`, the second part won't run if `x == 0`. This prevents errors like division by zero.

❖ 2.5.2. The if Statement

`if` runs a block of code only if a condition is true.


```
#include <iostream>

int main() {
    int age = 20;

    if (age >= 18) {
        std::cout << "Welcome! You are an adult.\n";
    }

    std::cout << "Done.\n";
    return 0;
}
```

❖ 2.5.3. if ... else

`else` provides an alternative when the condition is false.

```
int temp = 72;

if (temp > 85) {
    std::cout << "It's hot.\n";
} else {
    std::cout << "It's comfortable.\n";
}
```

❖ 2.5.4. else if (Multiple Branches)

Use `else if` to test more than one condition in order. The first true branch runs; the rest are skipped.

```
int score = 86;

if (score >= 90) {
    std::cout << "Grade: A\n";
} else if (score >= 80) {
    std::cout << "Grade: B\n";
} else if (score >= 70) {
    std::cout << "Grade: C\n";
} else if (score >= 60) {
    std::cout << "Grade: D\n";
} else {
    std::cout << "Grade: F\n";
}
```


Order Matters

Write the most specific or highest-threshold conditions first. In the grading example, if you checked `>= 80` before `>= 90`, an `A` score would be incorrectly caught by the `B` branch.

❖ 2.5.5. Blocks and Scope

Braces { } create a block. Variables declared inside a block exist only inside that block.

```
int number = 4;

if (number % 2 == 0) {
    std::string kind = "even";
    std::cout << number << " is " << kind << "\n";
}
// kind is not accessible here
```

❖ 2.5.6. Combining Conditions

Use logical operators to combine tests.

```
bool hasTicket = true;
bool isMember = false;

if (hasTicket || isMember) {
    std::cout << "You may enter.\n";
}

int x = 5;
if (x > 0 && x % 2 == 1) {
    std::cout << "x is a positive odd number.\n";
}

bool isReady = false;
if (!isReady) {
    std::cout << "Not ready yet.\n";
}
```


❖ 2.5.7. Comparing Strings and Characters

You can compare `char` and `std::string` values with `==` and `!=`.

```cpp
#include <string>

char grade = 'A';
if (grade == 'A') {
    std::cout << "Excellent!\n";
}

std::string color = "red";
if (color == "red") {
    std::cout << "Stop.\n";
} else if (color == "green") {
    std::cout << "Go.\n";
} else {
    std::cout << "Caution.\n";
}
```


Floating-Point Comparisons

Be careful comparing `float`/`double` with `==` because of tiny rounding differences. Prefer "close enough" checks using a small tolerance.

❖ 2.5.8. The switch Statement

`switch` selects one branch based on a single integral or enumeration value. It's a readable alternative to long `else if` chains when testing discrete cases.

```
int option = 2;

switch (option) {
    case 0:
        std::cout << "New game\n";
        break;
    case 1:
        std::cout << "Load game\n";
        break;
    case 2:
        std::cout << "Quit\n";
        break;
    default:
        std::cout << "Unknown option\n";
        break;
}
```


- **case**: a label to match against the `switch` value.
- **break**: exits the `switch`. Without it, execution "falls through" into the next case.
- **default**: runs when no case matches (optional but recommended).

Supported Types

`switch` works with integral types (like `int`, `char`, `bool`) and enums, which are explained below. It does not work with `float`/`double` or `std::string`.

Grouped Cases

Multiple `case` labels can share the same code. This is handy when several values should behave identically.

```
char ch = 'A';

switch (ch) {
    case 'A':
    case 'E':
    case 'I':
    case 'O':
    case 'U':
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        std::cout << "Vowel\n";
        break;
    default:
        std::cout << "Consonant or other\n";
        break;
}
```


Enums with switch

So far, you've used numbers or characters in a switch statement. But sometimes using plain numbers makes your code harder to read and easier to break. That's where **enums** come in.

What is an enum?

An **enum** (short for *enumeration*) is a special type that lets you create a small set of **named constant values**. Each name represents an underlying integer, starting from 0 by default unless you assign specific numbers.

For example:

```
enum class Direction { Up, Down, Left, Right }
```

Here, `Direction::Up` is `0`, `Direction::Down` is `1`, and so on.

Declaring enum variables

Once you've defined an enum, you can create variables of that type just like any other variable. Use the enum's name when declaring it, or use `auto` to let the compiler figure it out:

```
Direction dir = Direction::Up;
// or, using auto
auto dir = Direction::Up;
```

Later, when you get to Object-Oriented Programming, you'll see a similar pattern: using types with names like Menu or Player that group related values and behaviors together. Enums are a small, early example of that idea.

Why use enums?

1. **Readable**: You can write Menu::Quit instead of 3.
2. **Safe**: You can't accidentally mix up unrelated values like comparing a Menu with a Direction).
3. **Organized**: Related options live together under one name.

Enums make `switch` more descriptive and safer. Here's a rewritten version of the game `switch` from earlier:

```
enum class GameOption { NewGame, LoadGame, Quit }

GameOption option = GameOption::Quit;

switch (option) {
    case GameOption::NewGame:
        std::cout << "New game\n";
        break;
    case GameOption::LoadGame:
        std::cout << "Load game\n";
        break;
    case GameOption::Quit:
        std::cout << "Quit\n";
        break;
    default:
        std::cout << "Unknown option\n";
        break;
}
```


❖ 2.5.9. When to Use if vs. switch

- Use `if` for ranges, complex conditions, and expressions involving different variables.
- Use `switch` for a single value compared against many discrete options (especially `int`/`char`/enums).

❖ 2.5.10. Common Pitfalls to Avoid

- Using `=` instead of `==` in conditions. `=` assigns; `==` compares.
- Forgetting `break` in `switch` and falling through by accident.
- Comparing floating-point values with `==` directly. Prefer "close enough" checks.
- Missing braces around multi-statement `if`/`else` blocks.
- Writing overlapping or out-of-order `else if` conditions.

❖ 2.5.11. Putting It Together

Try out all of these examples in the below demo.

Demo 2.5: CoreSyntaxControlFlow/Demos/Conditionals.cpp

```
1.  #include <iostream>
2.  #include <string>
3.
4.  // Example enum for the final section
5.  enum class GameOption { NewGame, LoadGame, Quit };
6.
7.  int main() {
8.      // --- Booleans and Conditions ---
9.      int age = 20;
10.     bool isAdult = age >= 18;
11.     bool hasID = true;
12.     bool canEnter = isAdult && hasID;
13.
14.     std::cout << "Can enter? " << std::boolalpha << canEnter << "\n\n";
15.
16.     // Short-circuit example
17.     int x = 0;
18.     if (x != 0 && 10 / x > 2) { // second part won't run if x == 0
19.         std::cout << "No division by zero!\n";
20.     }
21.
22.     // --- The if Statement ---
23.     if (age >= 18) {
24.         std::cout << "Welcome! You are an adult.\n";
25.     }
26.     std::cout << "Done.\n\n";
27.
28.     // --- if ... else ---
29.     int temp = 72;
30.     if (temp > 85) {
31.         std::cout << "It's hot.\n";
32.     }
33.     else {
34.         std::cout << "It's comfortable.\n";
35.     }
36.     std::cout << "\n";
37.
38.     // --- else if (Multiple Branches) ---
39.     int score = 86;
40.     if (score >= 90) {
41.         std::cout << "Grade: A\n";
42.     }
43.     else if (score >= 80) {
44.         std::cout << "Grade: B\n";
```

Demo 2.5: CoreSyntaxControlFlow/Demos/Conditionals.cpp

```
45.       }
46.       else if (score >= 70) {
47.           std::cout << "Grade: C\n";
48.       }
49.       else if (score >= 60) {
50.           std::cout << "Grade: D\n";
51.       }
52.       else {
53.           std::cout << "Grade: F\n";
54.       }
55.       std::cout << "\n";
56.
57.       // --- Blocks and Scope ---
58.       int number = 4;
59.       if (number % 2 == 0) {
60.           std::string kind = "even";
61.           std::cout << number << " is " << kind << "\n";
62.       }
63.       // kind no longer exists here
64.       std::cout << "\n";
65.
66.       // --- Combining Conditions ---
67.       bool hasTicket = true;
68.       bool isMember = false;
69.
70.       if (hasTicket || isMember) {
71.           std::cout << "You may enter.\n";
72.       }
73.
74.       int y = 5;
75.       if (y > 0 && y % 2 == 1) {
76.           std::cout << "y is a positive odd number.\n";
77.       }
78.
79.       bool isReady = false;
80.       if (!isReady) {
81.           std::cout << "Not ready yet.\n";
82.       }
83.       std::cout << "\n";
84.
85.       // --- Comparing Strings and Characters ---
86.       char grade = 'A';
87.       if (grade == 'A') {
88.           std::cout << "Excellent!\n";
```

Demo 2.5: CoreSyntaxControlFlow/Demos/Conditionals.cpp

```
89.       }
90.
91.       std::string color = "red";
92.       if (color == "red") {
93.           std::cout << "Stop.\n";
94.       }
95.       else if (color == "green") {
96.           std::cout << "Go.\n";
97.       }
98.       else {
99.           std::cout << "Caution.\n";
100.      }
101.      std::cout << "\n";
102.
103.      // --- The switch Statement ---
104.      int option = 2;
105.      switch (option) {
106.          case 0:
107.              std::cout << "New game\n";
108.              break;
109.          case 1:
110.              std::cout << "Load game\n";
111.              break;
112.          case 2:
113.              std::cout << "Quit\n";
114.              break;
115.          default:
116.              std::cout << "Unknown option\n";
117.              break;
118.      }
119.      std::cout << "\n";
120.
121.      // --- Grouped Cases ---
122.      char ch = 'A';
123.      switch (ch) {
124.          case 'A':
125.          case 'E':
126.          case 'I':
127.          case 'O':
128.          case 'U':
129.          case 'a':
130.          case 'e':
131.          case 'i':
132.          case 'o':
```

Demo 2.5: CoreSyntaxControlFlow/Demos/Conditionals.cpp

```
133.         case 'u':
134.             std::cout << ch << " is a vowel.\n";
135.             break;
136.         default:
137.             std::cout << ch << " is a consonant or other character.\n";
138.             break;
139.     }
140.     std::cout << "\n";
141.
142.     // --- Enums with switch ---
143.     GameOption choice = GameOption::LoadGame; // could also use: auto choice =
          GameOption::LoadGame;
144.
145.     switch (choice) {
146.     case GameOption::NewGame:
147.         std::cout << "Starting a new game...\n";
148.         break;
149.     case GameOption::LoadGame:
150.         std::cout << "Loading saved game...\n";
151.         break;
152.     case GameOption::Quit:
153.         std::cout << "Exiting game.\n";
154.         break;
155.     default:
156.         std::cout << "Unknown option.\n";
157.         break;
158.     }
159.
160.     return 0;
161. }
```


With `if` and `switch`, your programs can react to different inputs and conditions, making them interactive and useful in real-world scenarios.

2.6. Loops: for, while, do-while

Loops let your program repeat actions without copying the same code over and over. You can count, process input repeatedly, or keep asking questions until a condition is met. In C++, the three fundamental loop types are `for`, `while`, and `do-while`. You'll also see how to stop a loop early with `break`, skip to the next iteration with `continue`, and iterate cleanly with range-based `for`.

❖ 2.6.1. The for Loop

Use a `for` loop when you know (or can compute) how many times you want to repeat something. It groups three parts in one line:

1. **Initialization**: runs once before the loop starts (often declares a counter).
2. **Condition**: checked before each iteration; the loop runs while this is `true`.
3. **Update**: runs after each iteration (often increments/decrements the counter).

```
// Anatomy: for (initialization; condition; update)
for (int i = 1; i <= 5; ++i) {
    std::cout << i << " ";
}
std::cout << "\n"; // Output: 1 2 3 4 5
```


Counting by a different step:

```
for (int i = 0; i < 10; i += 2) {
    std::cout << i << " ";
}
// Output: 0 2 4 6 8
```

Loop Variable Scope
A variable declared in the `for` initialization (e.g., `int i`) is only accessible inside the loop.

Summation Example

```
int n = 5;
int sum = 0;
for (int i = 1; i <= n; ++i) {
    sum += i; // 1 + 2 + 3 + 4 + 5
}
std::cout << "sum = " << sum << "\n"; // 15
```

< vs. <=

```
// Prints 0,1,2,3,4 (five numbers)
for (int i = 0; i < 5; ++i) { std::cout << i << " "; }

std::cout << "\n";

// Prints 0,1,2,3,4,5 (six numbers)
for (int i = 0; i <= 5; ++i) { std::cout << i << " "; }
```


❖ 2.6.2. The while Loop

Use a `while` loop when you want to repeat as long as a condition stays true, and you don't necessarily know how many times in advance.

Countdown Example

```
int count = 5;
while (count > 0) {
    std::cout << count << "\n";
    --count; // update: getting closer to the stopping point
}
std::cout << "Liftoff!\n";
```

Input Validation Example

```
int age = 0;
std::cout << "Enter age (1..120): " << std::flush;
std::cin >> age;

while (age < 1 || age > 120) { // loop while value is invalid
    std::cout << "Try again (1..120): " << std::flush;
    std::cin >> age;
}

std::cout << "Thanks! You entered " << age << ".\n";
```

❖ 2.6.3. The do-while Loop

`do-while` is like `while`, but the body runs **at least once** because the condition is checked at the end.


```
char again = 'n';
do {
    std::cout << "Performing an action...\n";
    std::cout << "Run again? (y/n): " << std::flush;
    std::cin >> again;
} while (again == 'y' || again == 'Y');

std::cout << "Done.\n";
```

This example will prompt the user with the question "Run again?" as long as they continue to tell it do so (by inputting 'y'). However, even before the user has the chance to input anything, it will run one time.

❖ 2.6.4. Stopping or Skipping: break and continue

- `break`: exits the loop immediately.
- `continue`: skips the rest of the current iteration and moves to the next one.

```
// Find the first multiple of 7 between 1 and 100
for (int i = 1; i <= 100; ++i) {
    if (i % 7 == 0) {
        std::cout << "First multiple of 7: " << i << "\n";
        break; // stop the loop early
    }
}
// Output: First multiple of 7: 7
```

```
// Print only even numbers from 1 to 10
for (int i = 1; i <= 10; ++i) {
    if (i % 2 == 1) {
        continue; // skip odd numbers
    }
    std::cout << i << " ";
}
// Output: 2 4 6 8 10
```

❖ 2.6.5. Range-Based for (Looping Over a Collection)

A range-based `for` loop is a simpler way to visit each element in a sequence. You'll use it often with strings, arrays, and other containers.

Here is an example of looping through the characters in a string. You will learn about other containers later in the course.

```
#include <string>

// Iterate characters in a string
std::string name = "Ada";
for (char ch : name) {
    std::cout << "[" << ch << "]";
}
// Output: [A][d][a]
```

❖ 2.6.6. Nested Loops

A loop inside another loop is a *nested loop*. The inner loop runs completely for each iteration of the outer loop.

```
// Print a 3x5 rectangle of '*'
int rows = 3;
int cols = 5;

for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) {
        std::cout << '*';
    }
    std::cout << "\n";
}
```

Infinite Loops (and How to Avoid Them)

A loop runs forever if its condition never becomes `false`. Common causes: a missing update (like forgetting `++i`) or a condition that never changes. If you really want an intentional infinite loop, use `while (true)` and include a `break` when it's time to stop.


```
// Intentional infinite loop with a clean exit
int attempts = 0;
while (true) {
    std::cout << "Attempt " << attempts << "\n";
    if (++attempts == 3) {
        break; // exit after 3 attempts
    }
}
```

❖ 2.6.7. Putting It Together

Demo 2.6: CoreSyntaxControlFlow/Demos/Loops.cpp

```cpp
#include <iostream>
#include <string>

int main() {
    // --- The for Loop ---
    std::cout << "--- for Loop ---\n";

    // Count from 1 to 5
    for (int i = 1; i <= 5; ++i) {
        std::cout << i << " ";
    }
    std::cout << "\n\n";

    // Count by 2s
    for (int i = 0; i < 10; i += 2) {
        std::cout << i << " ";
    }
    std::cout << "\n\n";

    // Summing numbers from 1 to n
    int n = 5;
    int sum = 0;
    for (int i = 1; i <= n; ++i) {
        sum += i;
    }
    std::cout << "Sum of 1..5 = " << sum << "\n\n";

    // Compare < vs <=
    for (int i = 0; i < 5; ++i) {
        std::cout << i << " ";
    }
    std::cout << "\n";
    for (int i = 0; i <= 5; ++i) {
        std::cout << i << " ";
    }
    std::cout << "\n\n";

    // --- The while Loop ---
    std::cout << "--- while Loop ---\n";

    int count = 5;
    while (count > 0) {
        std::cout << count << "\n";
        --count;
```

Demo 2.6: CoreSyntaxControlFlow/Demos/Loops.cpp

```
45.       }
46.       std::cout << "Liftoff!\n\n";
47.
48.       // Input validation example
49.       int age = 0;
50.       std::cout << "Enter age (1..120): " << std::flush;
51.       std::cin >> age;
52.
53.       while (age < 1 || age > 120) {
54.           std::cout << "Try again (1..120): " << std::flush;
55.           std::cin >> age;
56.       }
57.       std::cout << "Thanks! You entered " << age << ".\n\n";
58.
59.       // --- The do-while Loop ---
60.       std::cout << "--- do-while Loop ---\n";
61.
62.       char again = 'n';
63.       do {
64.           std::cout << "Performing an action...\n";
65.           std::cout << "Run again? (y/n): " << std::flush;
66.           std::cin >> again;
67.       } while (again == 'y' || again == 'Y');
68.
69.       std::cout << "Done.\n\n";
70.
71.       // --- Stopping or Skipping: break and continue ---
72.       std::cout << "--- break and continue ---\n";
73.
74.       // Find the first multiple of 7
75.       for (int i = 1; i <= 100; ++i) {
76.           if (i % 7 == 0) {
77.               std::cout << "First multiple of 7: " << i << "\n";
78.               break; // stop early
79.           }
80.       }
81.
82.       // Print even numbers using continue
83.       for (int i = 1; i <= 10; ++i) {
84.           if (i % 2 == 1) {
85.               continue; // skip odd numbers
86.           }
87.           std::cout << i << " ";
88.       }
```

Demo 2.6: CoreSyntaxControlFlow/Demos/Loops.cpp

```
89.       std::cout << "\n\n";
90.
91.       // --- Range-Based for Loop ---
92.       std::cout << "--- Range-Based for Loop ---\n";
93.
94.       std::string name = "Ada";
95.       for (char ch : name) {
96.           std::cout << "[" << ch << "]";
97.       }
98.       std::cout << "\n\n";
99.
100.      // --- Nested Loops ---
101.      std::cout << "--- Nested Loops ---\n";
102.
103.      int rows = 3;
104.      int cols = 5;
105.
106.      for (int r = 0; r < rows; ++r) {
107.          for (int c = 0; c < cols; ++c) {
108.              std::cout << '*';
109.          }
110.          std::cout << "\n";
111.      }
112.      std::cout << "\n";
113.
114.      // --- Infinite Loop with Exit ---
115.      std::cout << "--- Infinite Loop with break ---\n";
116.
117.      int attempts = 0;
118.      while (true) {
119.          std::cout << "Attempt " << attempts << "\n";
120.          if (++attempts == 3) {
121.              break; // exit after 3 tries
122.          }
123.      }
124.
125.      std::cout << "Finished all demos.\n";
126.      return 0;
127.  }
```


❖ 2.6.8. Common Pitfalls to Avoid

- Forgetting the update step in a loop (e.g., missing `++i`) leading to an infinite loop.

- Using the wrong condition (`<` vs. `<=`) and getting off-by-one results.
- Changing the loop variable in surprising ways inside the body; keep updates simple and clear.
- Not handling user input that doesn't meet expectations (like values out of range); add checks and repeat prompts as needed.

With these loop tools, you can repeat work efficiently, validate input reliably, and structure programs that respond to real-world tasks.

Exercise 2: Control Flow Practice

15 to 30 minutes

In this exercise, you will build a small console program that manages simple bank account actions. You will present a menu, validate input, and update a running balance.

❖ E2.1. Requirements

1. Create a menu-driven program that supports these options:
    - Deposit (`d`)
    - Withdraw (`w`)
    - Show Balance (`b`)
    - Quit (`q`)
2. Start balance at `$0.00`.
3. Deposit and withdrawal amounts must be **positive numbers**. If the user enters a zero or negative number, keep prompting until a valid value is entered.
4. Deny any withdrawal that exceeds the current balance.
5. Keep showing the menu until the user chooses Quit.
6. All input should be handled using `std::string` and `std::getline()`, with numeric values converted using `std::stod()`.

❖ E2.2. Step-by-step instructions

1. Create a new file: `CoreSyntaxControlFlow/Exercises/BankingMenu.cpp`.
2. Include the necessary headers: `#include <iostream>`, `#include <string>`, and `#include <format>`.
3. In `main()`, declare:
    A. `const double START_BALANCE = 0.0;`
    B. `double balance` initialized to this starting balance
    C. `std::string userInput` to store menu selections and numeric input
4. Use a **do-while** loop to:

A. print the menu options

B. read the user's choice using `std::getline()`

C. ignore empty input lines

D. use `switch(userInput[0])` to handle the chosen option

5. In the **Deposit** and **Withdraw** cases:

   A. prompt the user for an amount

   B. read it using `std::getline()`

   C. convert it to a number using `std::stod()`

   D. use a `while` loop to ensure the amount is **positive**

   E. update the balance using `+=` for deposits and `-=` for withdrawals

6. For withdrawals, check if the amount is greater than the current balance and deny the transaction if so.

7. Exit the loop when the user selects `'q'` or `'Q'`.

8. Compile and test your program to ensure all menu options and validations work correctly.

Solution: CoreSyntaxControlFlow/Solutions/BankingMenu.cpp

```cpp
#include <iostream>
#include <format>
#include <string>

int main() {
    const double START_BALANCE = 0.0; // Initial account balance
    double balance = START_BALANCE;   // User's account balance
    std::string userInput;            // User's menu choice

    do {
        // Display menu options
        std::cout << "\n Deposit (d)\n";
        std::cout << " Withdraw (w)\n";
        std::cout << " Balance (b)\n";
        std::cout << " Quit (q)\n\n";
        std::cout << "Enter choice: ";
        std::getline(std::cin, userInput);

        if (userInput.empty()) continue;  // ignore blank lines

        switch (userInput[0]) {
            // Handle deposit
            case 'd':
            case 'D': {
                double amount;
                std::cout << "Enter deposit amount: ";
                std::getline(std::cin, userInput);
                amount = std::stod(userInput);

                // Use while loop for validation
                while (amount <= 0) {
                    std::cout << "Amount must be positive. Try again: ";
                    std::getline(std::cin, userInput);
                    amount = std::stod(userInput);
                }

                balance += amount;
                std::cout << std::format("Deposited ${:.2f}\n", amount);
                break;
            }
            // Handle withdrawal
            case 'w':
            case 'W': {
                double amount;
```

Solution: CoreSyntaxControlFlow/Solutions/BankingMenu.cpp

```
                std::cout << "Enter withdrawal amount: ";
                std::getline(std::cin, userInput);
                amount = std::stod(userInput);

                while (amount <= 0) {
                    std::cout << "Amount must be positive. Try again: ";
                    std::getline(std::cin, userInput);
                    amount = std::stod(userInput);
                }

                if (amount > balance) {
                    std::cout << "Insufficient funds. Withdrawal denied.\n";
                } else {
                    balance -= amount;
                    std::cout << std::format("Withdrew ${:.2f}\n", amount);
                }
                break;
            }
            // Display balance
            case 'b':
            case 'B':
               std::cout << std::format("Current balance: ${:.2f}\n", balance);
                break;
            // Handle quit
            case 'q':
            case 'Q':
                std::cout << std::format("Final balance: ${:.2f}\n", balance);
                break;
            // Handle invalid input
            default:
              std::cout << "Invalid selection. Please choose a valid option.\n";
        }

    } while (userInput[0] != 'q' && userInput[0] != 'Q');

    return 0;
}
```


Conclusion

In this lesson on Core Syntax and Control Flow, we've explored the fundamental building blocks that constitute programming languages. By understanding syntax, we gain insight into the structure and

rules that define how programs should be written. From simple statements to complex expressions, syntax forms the foundation of code clarity and functionality.

The exploration of control flow has introduced us to the dynamic ways in which programs can maneuver through different pathways. Through constructs like loops, conditionals, and branches, control flow provides the means to implement logic and decision-making within our code. This enables the creation of responsive and adaptable software solutions.

Together, core syntax and control flow empower programmers to write efficient, effective, and maintainable code. They are not just basic concepts but essential competencies in the toolkit of every developer, forming the basis for more advanced programming paradigms and applications.

As you continue your journey in programming, remember that mastering these core elements is crucial for unlocking the potential of more complex and innovative projects. Embrace the practice and application of these concepts to enhance your problem-solving capabilities and creative coding endeavors.

LESSON 3

Functions and Functional Features

Topics Covered

☑ Function definition syntax.

☑ Function overloading and default arguments.

☑ Variable scope rules.

☑ Lambda expressions.

☑ Compile-time computation with constexpr and consteval.

☑ Error handling strategies.

Introduction

In this lesson, you'll learn the fundamentals of working with functions in C++, including how to declare, define, and call them, return values, and pass parameters by value or reference. You'll also explore ways to make your code cleaner with function overloading and default parameters, and understand how scope and lifetime control where variables can be used and how long they exist.

You'll then discover modern C++ features like lambdas and generics for writing concise, reusable code, and learn the difference between consteval and constexpr for compile-time calculations. You'll also practice safe error handling with try, catch, and standard exceptions. Finally, you'll apply these concepts by refactoring a simple banking menu, creating programs that are clearer, safer, and easier to maintain.

3.1. Defining and Using Functions

In this activity, you'll explore the basics of C++ functions - what they are, how to define and call them, and how they handle parameters (by value or by reference). Each subsection includes a short code example to help you see how these concepts work in practice.

❖ 3.1.1. What Is a Function?

A function in C++ is a reusable block of code that performs certain actions. It lets you organize your code into small, reusable parts.

By using functions, you can write code once and reuse it whenever you need it. This saves time and makes your programs easier to build, debug, and maintain. Common uses of functions include:

- Performing calculations or conversions.
- Validating input data.
- Performing mathematical or statistical operations.

You've already seen one important function: `main()`, which is required for any console-based program (programs run in a terminal view or command prompt).

❖ 3.1.2. Declaring a Function

Before you can call or use a function, it must be declared. A function declaration tells the compiler three things:

- What type of value the function returns.
- The name of the function.
- The parameters passed to the function.

Here is an example:

```
int addInts(int a, int b); //declaration
```

This line says that `addInts` is a function that takes two integers (`a` and `b`), and returns an `int` (integer) data type.

Declarations

In C++, function declarations are often stored in header files (files ending in `.h`). The header file serves as documentation - it shows *how* to use the function, but doesn't include the implementation.

Here is an example of including a header file in your main program:

```
#include "addInts.h" // Includes the local header file "addInts.h" so its declarations
can be used
```

❖ 3.1.3. Defining a Function

The function definition contains the code that runs when the function is called. A function definition tells the compiler what the function does. Here is an example of a function definition in C++:


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

This function takes two integers as parameters (a and b), adds them together, and returns an integer.

❖ 3.1.4. Function with Return Value

Functions can return values using the `return` keyword. Once the statement containing this keyword runs, the value returns to the caller, and the function ends. Following is an example of using a return value:

```
int result = addInts(10,20);
std::cout << "Result from the function:" << result;
```

Here, the `addInts(10, 20)` function runs and returns the sum. The sum is stored in `result` and printed to the console.

❖ 3.1.5. Passing Parameters by Value

In C++, we can pass parameters in two ways. One of them is by value. When passing parameters by value, the function gets a copy of the variable. So, any changes made inside the function affect only the copy, not the original variable. Note that passing by value takes up more memory because two memory locations are made for the same variable.

Here is an example of passing a parameter by value:

```
#include <iostream>
void addOneByValue(int n) { // n is a copy
    n = n + 1; // modifies only the copy
}
int main() {
    int x = 5;
    std::cout << "Original variable before function call:" << x << "\n"; // Output=5
    addOneByValue(x); // function call
    std::cout << "Original variable after the variable call:" << x << "\n"; // Output=5
}
```


❖ 3.1.6. Passing Parameters by Reference

Another way of passing parameters to a function in C++ is by reference. This method uses an ampersand (&) in the parameter type. The function works with the original variable, so any changes made inside the function after the caller's variable. Below is an example of passing parameters by reference:

```
#include <iostream>
void addOneByReference(int& n) { // n refers to the caller's variable
    n = n + 1; // modifies the original variable
}
int main() {
    int x = 5;
    std::cout << "Original variable before function call:" << x << "\n" ; // Output = 6

    addOneByReference(x); // function call
    std::cout << "Original variable after function call:" << x << "\n" ; // Output = 6
}
```

The main benefit of passing a parameter by reference is that the function can change the original variable, and those changes are visible to the caller. Normally, a function can only return one value

using the return statement. But if you need to update or send back multiple values, passing parameters by reference is a convenient way to do it.

Demo 3.1: FunctionsFunctionalFeatures/Demos/Functions.cpp

```
1.   #include <iostream>
2.
3.   // Function declarations
4.   int add(int a, int b);              // returns a value
5.   void addOneByValue(int n);          // gets a copy
6.   void addOneByReference(int& n);     // works on the original variable
7.
8.   // Function definitions
9.   int add(int a, int b) {
10.      return a + b;   // send a value back to the caller
11.  }
12.
13.  void addOneByValue(int n) {         // n = copy of caller's variable
14.      n = n + 1;                      // only the copy is changed
15.  }
16.
17.  void addOneByReference(int& n) {    // n refers to caller's variable
18.      n = n + 1;                      // original variable changes
19.  }
20.
21.  int main() {
22.
23.      // Calling a function with return value
24.      int sum = add(3, 4);
25.      std::cout << "add(3, 4) = " << sum << "\n";
26.
27.      // Passing by value
28.      int a = 5;
29.      addOneByValue(a);               // a does NOT change
30.      std::cout << "After addOneByValue: " << a << "\n";
31.
32.      // Passing by reference
33.      int b = 5;
34.      addOneByReference(b);           // b DOES change
35.      std::cout << "After addOneByReference: " << b << "\n";
36.
37.      return 0;
38.  }
```

3.2. Overloading and Default Parameters

After learning how to define and use functions, it's important to understand some advanced features that make your code more flexible and readable.

C++ allows you to:

- Use the same function name for different tasks
- Provide default values for function arguments.

These features are called **function overloading** and **default parameters**.

❖ 3.2.1. Function Overloading

Function overloading lets you define multiple functions with the same name but with different parameters. The compiler automatically chooses the correct version depending on the arguments you provide.

Why It's Useful:

- Makes your code cleaner and easier to read.
- Avoids creating multiple functions with different names for similar tasks.

Rules for Function Overloading:

- Functions must have the same name.
- Functions must differ in number, types, or order of parameters.
- The return type alone cannot distinguish overloaded functions.

```
#include <iostream>

void print(int number) { // Overloaded print function: handles integer input
    std::cout << "Integer: " << number << "\n";
}

void print(string text) { // Overloaded print function: handles string input
    std::cout << "String: " << text << "\n";
}

int main() {
    print(10);        // Calls int version
    print("Hello");   // Calls string version
    return 0;
}
```


❖ 3.2.2. Default Parameters

Default parameters let you assign a default value to function arguments. If the caller does not provide that argument, the function uses the default value.

Why It's Useful:

- Reduces the need for multiple overloaded functions.
- Simplifies function calls.

Rules for Default Parameters:

- Defaults must be specified in the function declaration or definition.
- Default parameters must appear at the end of the parameter list.

```
#include <iostream>

void greet(string name = "Guest") { // Function with a default argument
   std::cout << "Hello, " << name << "!" << "\n";
}

int main() {
    greet("Alice");  // Prints: Hello, Alice!
    greet();         // Prints: Hello, Guest!
    return 0;
}
```

Demo 3.2: FunctionsFunctionalFeatures/Demos/OverloadingAndDefaultParam.cpp

```cpp
#include <iostream>
#include <string>


// Function Overloading
// Same name, different parameters
void show(int number) {
std::cout << "You passed an integer: " << number << "\n";
}

void show(const std::string& text) {
std::cout << "You passed a string: " << text << "\n";
}

// Default Parameter
// If 'name' is not provided, "Guest" is used
void greet(std::string name = "Guest") {
std::cout << "Hello, " << name << "!\n";
}

int main() {

    // --- Overloading examples ---
    show(42);           // calls the int version
    show("Hello");      // calls the string version

    // --- Default parameter examples ---
    greet("Alice");     // uses provided argument
    greet();            // uses default "Guest"

    return 0;
}
```

3.3. Scope and Lifetime

Now that you understand how functions work in C++, it's time to explore how variables behave both inside and outside those functions.

Scope tells you *where* in your program a variable can be accessed by its name, while **lifetime** describes *how long* that variable exists in memory and remains valid.

❖ 3.3.1. Understanding Scope

Scope defines the specific area of your code where a variable is visible and can be used. This is important because the scope:

- helps control how long data should exist.
- limits where variables can be accessed.
- prevents issues like accidentally using the wrong variable name.
- helps your program stay memory-efficient.

C++ has different types of scope:

Local Scope

Variables declared inside a function or braces `{}` are said to have local scope. This means they can only be used within that specific function or block.

```
void greet() {
    string name = "Alice";  // local to the function
    std::cout << name << "\n";
}
// 'name' cannot be used here
```

Once the function finishes, the variable is no longer accessible.

Block Scope

Block scope is a more specific form of local scope. A variable declared inside a smaller block—such as an `if`, `for`, or `while` loop—exists only inside that block.

```
for (int i = 0; i < 5; i++) {
    // 'i' is only accessible inside this loop
}
// 'i' is not accessible here
```

This prevents variables from interfering with each other across different parts of the program.

Global Scope

A variable declared outside of all functions has global scope. Global variables can be accessed from any function in the same file, and even from other files if you use the `extern` keyword.

```
int score = 100;  // global variable

void display() {
    std::cout << score;
}
```

Namespace Scope

Variables declared inside a `namespace` belong to that namespace. You can access them by using the namespace's name or a `using` directive. Namespaces are especially useful in large projects where many variables and functions might share the same names.

```
namespace Config {     // Namespace grouping related variables/functions under "Config"

    int version = 2;               // Variable stored inside the Config namespace
}

int main() {
    std::cout << Config::version; // Accesses the version variable using the namespace
scope
}
```

Lifetime describes how long a variable actually exists in memory while your program is running. A variable's lifetime ends when the variable goes **out of scope**. For example, a variable defined in a function goes out of scope when the function terminates. A global go out of scope when the `main` function terminates.

Local Variables

Local variables are declared inside functions or blocks.

- Created when the block or function is entered.
- Destroyed when the block or function ends.

```
void greet() {
    string name = "Zoya";
    std::cout << "Hello, " << name << std::endl;
}  // 'name' is destroyed here
```

Every time `greet()` is called, a new `name` variable is created, used, and then destroyed. It does not keep its value between calls.

Global Variables

Global variables are declared outside all functions.

- Created when the program starts.
- Destroyed when the program ends.

```
int count = 0;           // Global variable accessible from anywhere in this file

int main() {
    count++;             // Increments the global variable by 1
    std::cout << count;  // Prints the updated value
}
```

Static Variables

Static variables inside functions are declared using the `static` keyword.

- Initialized only once.
- Keep their value between function calls.

```
void increment() {              // Function that keeps a persistent internal counter
    static int count = 0;       // Initialized once and retains its value between calls

    count++;                    // Increments the stored count
    std::cout << count << "\n"; // Prints the current count
}
```


Static variables have local scope but global lifetime. They are useful when you need a function to remember information from previous calls.

❖ 3.3.2. Best Practices

- Prefer the smallest scope possible so variables are only visible where they are needed.
- Use clear, descriptive variable names to avoid confusion and name shadowing.
- Keep global state minimal; pass values explicitly rather than relying on shared variables.

- Initialize variables close to where they are used to improve clarity.

Demo 3.3: FunctionsFunctionalFeatures/Demos/ScopeAndLifetime.cpp

```cpp
#include <iostream>

// Global scope & lifetime
// Exists for the entire program
int globalCount = 0;

// Namespace scope
namespace Config {
    int version = 1;
}

// Function demonstrating local & static variables
void demoFunction() {
    int localVar = 10;          // local scope, destroyed each time
    static int staticVar = 0;   // static lifetime, keeps value between calls

    staticVar++; // remembers previous value
    std::cout << "localVar: " << localVar << "\n";
    std::cout << "staticVar: " << staticVar << "\n";
}

int main() {

    // Local scope
    int x = 5;   // exists only inside main()
    std::cout << "x: " << x << "\n";


    // Block scope
    if (true) {
        int y = 20; // exists only inside this block
        std::cout << "y inside block: " << y << "\n";
    }
    // y is not accessible here

    // Using global variable
    globalCount++;
    std::cout << "globalCount: " << globalCount << "\n";

    // Namespace variable
    std::cout << "Config::version: " << Config::version << "\n";

    // Demonstrate local vs static lifetime
    demoFunction();
```

Demo 3.3: FunctionsFunctionalFeatures/Demos/ScopeAndLifetime.cpp

```
45.       demoFunction();
46.       demoFunction(); // staticVar keeps increasing
47.
48.       return 0;
49.   }
```


3.4. Lambdas and Generics

Modern C++ gives you features that make your code shorter, cleaner, and more reusable. Two of the most powerful features are lambdas and generics. They may sound technical, but once you understand the basics, they're easy to use.

❖ 3.4.1. Lambdas

A lambda expression is a small, anonymous function that can be written directly in the code, useful for single-use functions without needing a separate name or declaration. Think of it as a *quick function* you write on the spot.

Basically, lambdas are used to encapsulate a few lines of code that are passed to algorithms or asynchronous functions.

Reasons to use lambdas:

- No need to create a separate function.
- No need to write a class (function objects) like the older C++.
- Keeps logic close to where it is used.
- Great for short, simple operations.

Lambda Structure

```
[capture] (parameters) -> return_type { body };
```

Here,

- **Capture**: How the lambda accesses outside variables.
- **Parameters**: Values you pass in.
- **Return type**: Type of value the lambda produces. This is optional as it can usually be deduced automatically.

Basic Lambda Example

```
auto add = [](int a, int b) { // Creates an anonymous function that adds two integers
    return a + b;
};

std::cout << add(3, 5);   // Calls the lambda with 3 and 5, then prints the result = 8
```

Capture List

The capture list ([ ]) controls which variables from outside the lambda the function may use. Common capture modes:

- `[ ]` - capture nothing.
- `[x]` - capture `x` by value (copy).
- `[&x]` - capture `x` by reference.
- `[=]` - capture all used local variables by value.
- `[&]` - capture all used variables by reference.
- `[=, &x]` - capture everything by value, but capture `x` by reference.
- `[&, x]` - capture everything by reference, but capture `x` by value.
- `[this]` - capture the current object (useful in classes).

These rules determine which variables the lambda can "see," and whether it uses copies or references.

Parameter List

The parameter list (( )) describes the input the lambda receives.

- `(x)` - one parameter.
- `(a, b)` - multiple parameters.

- `()` - no parameters (optional, but better for higher readability).
- `(auto x)` - automatic type deduction.

Return Type

The return type is mostly deducted automatically and may be omitted if the lambda returns a single obvious type or nothing. We can specify it using the arrow syntax `-> type` when you want to be explicit or when deduction is unclear.

```
auto add = [](int a, int b) -> int { // Lambda expression with explicit return type 'int'

    return a + b;  // Returns the sum of a and b
};
```

❖ 3.4.2. Generics

Generics is the idea of allowing types to be a parameter to methods, classes, and interfaces. In C++, generics are implemented through templates. **Templates** enable writing code that can work with different data types without requiring separate implementations for each type.

Reasons to use generics (templates):

- Code reusability and efficiency.
- More flexible and adaptable code.
- Catches type-related errors early.

Template Parameters

Template uses type or non-type values as parameters, which are specified within angular brackets `< >`.

- Type values: We declare using `typename` or `class` (e.g., `template <typename T>`), where `T` serves as a placeholder for any data type.
- Non-type parameters: These allow passing constant values as template arguments (e.g., `template <int N>`).

A type value is a data type passed to a template, like `int` or `double`, whereas a non-type value is a constant value passed to a template, like `7` or `3.14`.

Generic Functions (Function Templates)

Generic functions are defined with template parameters, allowing them to operate on various data types. The compiler generates specialized versions of the function for each distinct type used.

```
template <typename T>
T add(T a, T b) {
     return a + b;
}
```

- Here, we declare a template parameter `T`.
- Then we use `T` as the type for the two parameters of the `add` function.
- This means our function can accept any two values as long as they are the same type, so for example, it could accept two ints or two doubles, and the function would still work.

❖ 3.4.3. Combining Lambdas and Generics

You can write generic functions that accept lambdas for flexible, reusable behavior:

```
template <typename T, typename F>
void apply(T value, F func) {
    std::cout << func(value) << "\n";
}

apply(5, [](int x){ return x * 2; }); // prints 10
```

In this example, we have created a generic `apply` function that can apply any function (or lambda) to any value and then print out the result.

Try out these examples in the demo below!

Demo 3.4: FunctionsFunctionalFeatures/Demos/Lambdas.cpp

```
1.   #include <iostream>
2.
3.   // Generic function (template)
4.   // Works with any type T and any callable F
5.
6.   template <typename T, typename F>
7.   void apply(T value, F func) {
8.   std::cout << "Result: " << func(value) << "\n";
9.   }
10.
11.  int main() {
12.
13.      // Basic lambda
14.      auto add = [](int a, int b) {
15.          return a + b;
16.      };
17.
18.      std::cout << "add(3, 5) = " << add(3, 5) << "\n";
19.
20.      // Lambda with capture
21.      int factor = 2;
22.
23.      auto multiply = [factor](int x) {   // captures factor by value
24.          return x * factor;
25.      };
26.
27.      std::cout << "multiply(10) = " << multiply(10) << "\n";
28.
29.      // Using a lambda with a generic function
30.      apply(5, [](int x) { return x * 3; });   // prints 15
31.
32.      return 0;
33.  }
```


3.5. consteval and constexpr

`consteval` functions let the compiler calculate values at compile time, instead of waiting for runtime. Using `consteval` functions can potentially improve runtime performance.

`constexpr` functions also let the compiler calculate values at runtime, but in addition permit runtime evaluation as well.

In this activity you will learn about these important function types and when to use them.

❖ 3.5.1. consteval

You can use `consteval` when a function must be evaluated at compile time. Attempting to call the function at runtime causes a compile-time error.

First, we will create `consteval` function:

```
consteval int triple(int x) {
    return x * 3;
}
```

Note the following:

1. `consteval` marks the function as a compile-time function.
2. The function has no dependency on variables except the parameter `x`.

Now we will call the function twice, first with a **literal** integer and then with a constant:

```
int i = triple(6);
std::cout << "triple(6) = " << i << " (computed at compile time)\n";
const int I = 6;
i = triple(I);
std::cout << "triple(I) = " << i << " (computed at compile time using a constant)\n";
```

In both cases the compiler is able to do the arithmetic stored in the function to obtain the result.

You **cannot** pass a variable to a `consteval` function without getting an error. For example, the following line of code will produce a compile error:

```
int r;
i = triple(r);  // compile error!
```

❖ 3.5.2. constexpr

A `constexpr` function can be computed at either compile time or runtime.

If the parameters passed to the function are literals or constants and the function has no dependencies on variables then the compiler can compute the result as is the case with `consteval`.

Otherwise, the result of the function will be computed at runtime.

Here is the constexpr function:

```
constexpr int square(int x) {
    return x * x;
}
```

The `constexpr` keyword identifies the function as a compile-time or runtime function.

We can call this function with literal, constant, or variable parameters:

```
 // call constexpr function with literal
 int j = square(7);
 std::cout << "square(7) = " << j << " (constexpr computed at compile time)\n";


 // call constexpr function with constant
 const int J = 7;
 j = square(J);
 std::cout << "square(J) = " << j << " (constexpr computed at compile time using a con  ↵↵
stant)\n";

 // call constexpr function with a variable
 int y = 7;
 int k = square(y);           // pass a variable to the function
 std::cout << "square(y) = " << k << " (constexpr computed at runtime)\n";
```

Items of interest

1. The result of the `square` function can be determined by the compiler when a **literal** or **constant** is passed to the function. This behavior is identical to `consteval`.
2. The `square` function can also be called using a **variable**. The result will be determined at runtime.

Using consteval and constexpr

consteval is tenable if all of the parameters passed to a function are **hard-coded**. Some examples are

- Number of states in the Union
- Number of counties in a state
- The zip code of a city
- Annual Percentage Rate (APR) for a credit card

If the function at times must rely on **variables** then mark the function as `constexpr`.

If you do not mark the function either as `consteval` or `constexpr`, then the function results will **always** be evaluated at runtime.

Try out these examples using the following demo file.

Demo 3.5: FunctionsFunctionalFeatures/Demos/constevalAndconstexpr.cpp

```
1.   #include <iostream>
2.
3.   // consteval example
4.   // This function MUST run at compile time.
5.   consteval int triple(int x) {
6.       return x * 3;
7.   }
8.
9.   // constexpr example
10.  // This function CAN run at compile time, but can also run at runtime.
11.  constexpr int square(int x) {
12.      return x * x;
13.  }
14.
15.  int main() {
16.      // consteval usage
17.      int i = triple(6);
18.      std::cout << "triple(6) = " << i << " (consteval computed at compile time)\n";
19.      const int I = 6;
20.      i = triple(I);
21.      std::cout << "triple(I) = " << i << " (consteval computed at compile time
             using a constant)\n";
22.
23.      //int r;
24.      //i = triple(r);       // ERROR: triple must run at compile time. Uncomment ↵↵
             ing this will fail to compile.
25.
26.      // call constexpr function with a literal
27.      int j = square(7);
28.      std::cout << "square(7) = " << j << " (constexpr computed at compile time)\n";
29.      // call constexpr function with a constant
30.      const int J = 7;
31.      j = square(J);
32.      std::cout << "square(J) = " << j << " (constexpr computed at compile time
             using a constant)\n";
33.      // call constexpr function with a variable
34.      int y = 7;
35.      int k = square(y);           // pass a variable to the function
36.      std::cout << "square(y) = " << k << " (constexpr computed at runtime)\n";
37.
38.      return 0;
39.  }
```

3.6. Error Handling Techniques

Program errors are usually divided into two categories:

- **Logic errors** - An error that occurs when a program runs but generates unintended results, e.g., using the wrong operator.
- **Runtime errors** - An error that occurs during execution and may crash or stop the program, e.g., using deleted memory or accessing out-of-bound array elements.

In C++, the preferred method of reporting and handling these errors is to use exceptions. Exceptions provide a well-defined way for code that detects errors to pass the information up the call stack.

When writing code in C++, we can handle exceptions using the `try` and `catch` blocks, along with the `throw` keyword.

- `try` - code that may raise an exception.
- `throw` - throws an exception when an error is detected.
- `catch` - code that handles the exception thrown by the `throw` keyword.

Note: The `throw` statement is not mandatory, especially when using standard C++ exceptions.

❖ 3.6.1. Syntax for Error Handling in C++

Here is the basic syntax for handling exceptions in C++:

```
try {
  // Code that may raise an exception
  throw (argument); // Throws an exception with the given argument
}
catch (exception) { // Catches the thrown exception of type 'exception'
  // Code to handle exception
}
```

Here, we have put the code that might generate an exception within the `try` block. When an exception occurs, the `throw` statement throws an exception, which is caught by the `catch` block.

Each `try` block is followed by the `catch` block, and the `catch` block cannot be used without the `try` block.

❖ 3.6.2. Basic Example of Exception Handling

Below is the basic example of how to handle exceptions in C++:

```
try {
  std::string input = "qwerty";  // a non-integer string
  int number = std::stoi(input); // may throw an exception
  std::cout << "You entered: " << number << "\n";
}
catch (const std::invalid_argument& ex) {
  std::cout << "Invalid input! Message: " << ex.what() << "\n";
}
catch (const std::exception& ex) {
  std::cout << "Some other error occurred: " << ex.what() << "\n";
}
```


In this example, the `std::stoi` function is called. This function with attempt to convert the string representation of an **integer** into an `int` data type. If the input is not an integer, the function throws an `invalid_argument` error. The `catch` block catches that error and prints the result of calling the `what()` function of the error object, which is named `ex`. You can use any arbitrary name for the error object.

If we didn't catch the error, then the program would **terminate** when the error is thrown.

Note that you can use `std::stod` to convert the string representation of a double to a `double` data type. If the data is not a double, then `invalid_argument` will be thrown, as shown above to `std::stoi`.

❖ 3.6.3. Handling Any Type of Exceptions

In exception handling, we should be familiar with the types of exceptions that can occur due to the code in our `try` statement, so that we can use suitable `catch` parameters. Otherwise, the `try` and `catch` statement might not work properly.

In case the types of exceptions are unknown that can occur in our `try` block, C++ allows us to use the catch-all handler, i.e., `catch(...)`, to catch *any* exception.

```
try {
    // code
}
catch (...) {
    // code
}
```

This is how we can catch an exception without knowing its type.

When writing multiple `catch` statements, `catch (...) { }` should always be the final block in our `try...catch` statement because this block catches all possible exceptions and acts as the default `catch` block.

❖ 3.6.4. C++ Standard Exceptions

There are a number of standard exceptions in C++ that we can use in our exception handling. All these exceptions are defined in the `exception` header file and categorized into two main branches: `std::logic_error` and `std::runtime_error`. Some of them are as follows:

Exception	Description
`std::exception`	Parent class of all C++ exceptions
`std::invalid_argument`	Function received an invalid argument
`std::out_of_range`	The computational result is out of the allowed range
`std::bad_exception`	Thrown by `throw` when an exception specification allows it
`std::bad_alloc`	Dynamic memory allocation failed

Try some simple exception handling in the demo below!

Demo 3.6: FunctionsFunctionalFeatures/Demos/ExceptionHandling.cpp

```
1.   #include <iostream>
2.   #include <string>
3.   #include <exception>
4.
5.   int main() {
6.       try {
7.           std::string text = "abc";   // not a number
8.           std::cout << "Trying to convert text to a number...\n";
9.
10.          int number = std::stoi(text);   // may throw invalid_argument
11.          std::cout << "Number is: " << number << "\n";
12.      }
13.      catch (const std::invalid_argument& ex) {
14.          std::cout << "Invalid input! Cannot convert to number.\n";
15.          std::cout << "Message: " << ex.what() << "\n";
16.      }
17.      catch (...) {   // catches anything else
18.          std::cout << "An unknown error occurred.\n";
19.      }
20.
21.      std::cout << "Program continues safely.\n";
22.      return 0;
23.  }
```

Exercise 3: Function and Scope Practice

15 to 30 minutes

In this exercise, you will modify the Control Flow solution to use functions and to handle errors that might arise from validation of input.

❖ E3.1. Requirements

1. Add two functions to the Banking Menu app:
    A. `deposit` to add deposits to the account balance.
    B. `withdraw` to subtract withdrawals from the account balance.
2. Implement error handling in each function.

❖ E3.2. Step-by-step Instructions

1. Open `FunctionsFunctionalFeatures/Exercises/BankingMenu_Functions.cpp` as your starter file.
2. Create a function named `deposit` that takes the deposit amount as a double argument:
    - If the amount passed to the function is zero or less then throw an `runtime_error` exception with a descriptive message.
    - If the amount is valid, add the amount to the balance and return the balance.
3. Create a function named `withdraw` that takes the withdrawal amount as a double argument:
    - If the amount passed to the function is zero or less then throw an `runtime_error` exception with a descriptive message.
    - If the amount exceeds the balance then throw a `runtime_error` exception with an appropriate message.
    - If the amount is valid, subtract the amount from the balance and return the balance.
4. After calling either the deposit function or the withdraw function handle potential errors thrown by the functions:
    A. Display the message the function passed to constructor of runtime_error.
    B. Use the `what` function to display the message

5. Continue execution of the while loop even if errors are detected until the user enters `'Q'` or `'q'`.
6. Compile and test your program.

Solution: FunctionsFunctionalFeatures/Solutions/BankingMenu.cpp

```
1.  #include <iostream>
2.  #include <format>
3.  #include <stdexcept>
4.  #include <string>
5.
6.  const double START_BALANCE = 0.0;
7.  double balance = START_BALANCE;
8.  std::string userInput;
9.
10. double withdraw(double amount) {
11.     static int nbrOfWithdrawals = 1;
12.     if (amount > balance) {
13.         std::string error_msg = std::format("Withdrawal rejected: amount {:.2f}
              exceeds balance.\n", amount);
14.         throw std::runtime_error(error_msg);
15.     }
16.     else if (amount <= 0) {
17.         std::string error_msg = std::format("Withdrawal rejected: amount {:.2f}
              is less than or equal to zero.\n", amount);
18.         throw std::runtime_error(error_msg);
19.     }
20.     else {
21.         std::cout << std::format("Number of withdrawals is now {}\n",
             nbrOfWithdrawals);
22.         if (nbrOfWithdrawals > 3) {
23.             amount += 3;
24.         }
25.         nbrOfWithdrawals++;
26.         return balance -= amount;
27.     }
28. }
29.
30. double deposit(double amount) {
31.     if (amount <= 0) {
32.         std::string error_msg = std::format("Deposit rejected: amount {:.2f} is
              less than or equal to zero.\n", amount);
33.         throw std::runtime_error(error_msg);
34.     }
35.     else {
36.         return balance += amount;
37.     }
38. }
39.
40. int main() {
41.
```

Solution: FunctionsFunctionalFeatures/Solutions/BankingMenu.cpp

```
42.       do {
43.           // Display menu options
44.           std::cout << "\n Deposit (d)\n";
45.           std::cout << " Withdraw (w)\n";
46.           std::cout << " Balance (b)\n";
47.           std::cout << " Quit (q)\n\n";
48.           std::cout << "Enter choice: ";
49.           std::getline(std::cin, userInput);
50.
51.           if (userInput.empty()) continue;  // ignore blank lines
52.
53.           switch (userInput[0]) {
54.               // Handle deposit
55.               case 'd':
56.               case 'D': {
57.                   double amount;
58.                   std::cout << "Enter amount to deposit: ";
59.                   std::getline(std::cin, userInput);
60.                   try {
61.                       amount = std::stod(userInput);
62.                       balance = deposit(amount);
63.                       std::cout << std::format("Deposited ${:.2f}\n", amount);
64.                   }
65.                   catch (std::invalid_argument& ia) {
66.                    std::cout <<  "Deposit amount must be numeric, amount reject ↵↵
            ed.\n";
67.                   }
68.                   catch (std::runtime_error& re) {
69.                      std::cout << re.what();
70.                   }
71.                   break;
72.               }
73.               // Handle withdrawal
74.               case 'w':
75.               case 'W': {
76.                   double amount;
77.                   std::cout << "Enter withdrawal amount: ";
78.                   std::getline(std::cin, userInput);
79.                   try {
80.                       amount = std::stod(userInput);
81.                       balance = withdraw(amount);
82.                   }
83.                   catch (std::invalid_argument& ia) {
```

Solution: FunctionsFunctionalFeatures/Solutions/BankingMenu.cpp

```
84.                     std::cout <<  "Withdrawal amount must be numeric, amount re  ↵↵
              jected.\n";
85.                 }
86.                 catch (std::runtime_error& rte) {
87.                     std::cout << rte.what();
88.                 }
89.                 break;
90.             }
91.             // Display balance
92.             case 'b':
93.             case 'B':
94.                std::cout << std::format("Current balance: ${:.2f}\n", balance);
95.                 break;
96.             // Handle quit
97.             case 'q':
98.             case 'Q':
99.                 std::cout << std::format("Final balance: ${:.2f}\n", balance);
100.                break;
101.            // Handle invalid input
102.            default:
103.              std::cout << "Invalid selection. Please choose a valid option.\n";
104.        }
105.
106.    } while (userInput[0] != 'q' && userInput[0] != 'Q');
107.
108.    return 0;
109. }
```


Conclusion

In this lesson, you learned how to declare, define, and call functions, return values, and pass parameters by value or by reference. You also explored function overloading and default parameters to make your code simpler and easier to use. Additionally, you learned about variable scope (local, block, global, and namespace) and lifetime, including how automatic, static, and global variables work.

You practiced modern C++ features like lambdas and generic templates to write short, reusable code, and saw the difference between `consteval` and `constexpr` for compile-time calculations. You also learned how to handle errors safely using try, catch, and throw, including standard and catch-all exceptions. Finally, you applied these skills in a banking menu exercise, so your program runs safely and reliably.

LESSON 4
Arrays and Strings

Topics Covered

☑ Overview of C-style arrays and C-strings.

☑ Fixed-size arrays with std::array.

☑ Managing text with std::string.

☑ Non-owning string views with std::string_view.

☑ Formatting strings using std::format.

Introduction

In this lesson, you will explore how to work with arrays and strings in C++, starting with C-style arrays and strings, memory basics (bits, bytes, words), indexing, and the importance of the null terminator. You'll learn safer string operations (`strncpy()`, `strncat()`) and compare them with modern tools like `std::array` for fixed-size containers and `std::string` and `std::string_view` for flexible and efficient text handling, including safe access with `.at()` and helpful functions like `.size()`.

You'll also format readable output with `std::format` using placeholders, alignment, width, and precision. By the end, you'll be able to declare and iterate arrays, manage strings safely, and produce nicely formatted output.

EVALUATION COPY: Not to be used in class.

EVALUATION COPY: Not to be used in class.

4.1. C-style Arrays and Strings Overview

C++ is the **object-oriented form** of the C programming language. It is called object-oriented because it allows programmers to group related data and actions together into objects. Don't worry, we will explore that later.

The C language introduce arrays and strings (often called C-style arrays and C-style strings) as a way to help store and manage collections of data, such as numbers or letters.

❖ 4.1.1. Introduction to C-Style Arrays

A C-style array is a group of values that are all of the same type (**for example,** all integers or all characters). These values are stored next to each other in the computer's memory, which helps the computer find them quickly.

Arrays are defined using square brackets, and their size must be a constant integer expression. For instance:

```
int numbers[5]; // An array that can store 5 integers
```


This line creates an array named `numbers` with space for 5 integers. The valid positions (or indexes) are:

```
0,1,2,3,4
```

Memory: Bits, Bytes, and Words

Memory is the part of the computer that stores information while a program is running. It is also called RAM (Random Access Memory)**.** On most modern computers, one word of memory is usually 64 bits in size.

Arrays and strings use this memory space to store data. As the arrays are stored in memory, we need to decide how much space (how many elements) they will use before we start the program.

Bits, Bytes, and Words

A bit is the smallest piece of data a computer can store. It can be either `0` or `1`. Eight bits together make one byte, which can store a small piece of information (for example, one character like `'A'`).

When we say a word is 64 bits, it means the computer can process or move 8 bytes of data at one time. In simple terms, the "word size" tells us how much data the computer can handle in a single step. Modern 64-bit computers can handle more data and use more memory at once than older 32-bit systems.

Example: Checking size of data in C++

```
#include <iostream>
int main() {
    std::cout << "Size of char: " << sizeof(char) << " byte\n";
    std::cout << "Size of int: " << sizeof(int) << " bytes\n";
    return 0;
}
```

Explanation:

- The `sizeof` operator tells us how many bytes a data type uses in memory.
- 1 byte = 8 bits.
- On most 64-bit computers, a pointer (a variable that stores a memory address) usually takes 8 bytes, which equals 64 bits or one word of memory.

Note: We will cover memory in more detail in a later lesson.

Avoiding Errors When Using Arrays

C-style arrays do not check if you try to go outside their valid range. This means you can accidentally use memory that does not belong to your array. Doing this can cause **runtime errors** (errors that happen while your program is running) or even make the operating system crash.

So, we must be very careful when working with C-style arrays and strings to make sure we only use memory that has been properly assigned to our data.

Why Do We Use Arrays?

Arrays are useful when we need to store a fixed number of items of the same type. They make it easy to access each item using an index (a number that tells the position of each element).

However, C-style arrays:

- Have a fixed size (it cannot change while the program is running).

- Do not remember how many elements they contain.

Modern C++ provides safer options, like `std::vector`, which we will learn later.

```
int numbers[3] = {1, 2, 3};   // An array with 3 integers
```

Array indexes start at `0`, so the elements are `numbers[0]`, `numbers[1]`, and `numbers[2]`.

If you try to access `numbers[3]`, it will cause a runtime error or unexpected behavior.

Key Points to Remember:

- A C-style array stores values of the same type in one place.
- The size of the array is fixed when it is created.
- You must be careful not to access elements outside the array.
- Arrays are the building blocks for more advanced data structures in C++.

❖ 4.1.2. Exploring C-style Strings

A C-style string is simply an array of characters that ends with a special symbol called the null character written as `'\0'`.

This null character is very important because it tells the computer where the string ends. Without it, the program wouldn't know when to stop reading characters, which could lead to errors or unexpected results.

Here's an example of how to create a C-style string:

```
char greeting[] = "Hello"; // A C-style string
```

When you write a string in double quotes (`" "`), the null character `'\0'` is automatically added at the end. So, although `"Hello"` has 5 letters, the computer actually stores 6 characters: `H`, `e`, `l`, `l`, `o`, and `'\0'`. This means the total storage size of the string is the number of characters plus one extra space for the null character.

How C-Style Strings Are Stored in Memory

Let's look at how the string `"Hello"` is stored in memory:

```
Index:     0     1     2     3     4     5
Value:    'H'   'e'   'l'   'l'   'o'  '\0'
```

Each box represents one character stored in memory. The last box, `'\0'`, is the null character which tells the computer that the string ends here.

So, although `"Hello"` has 5 characters, the computer actually keeps 6 characters in memory (5 characters + 1 null terminator).

Quick Reminder: If you forget to include the null character when creating a C-style string manually, your program might continue reading memory beyond the string , which can cause errors or strange output.

❖ 4.1.3. Risks of Using C-style Arrays and Strings

As mentioned earlier, C-style arrays and C-style strings can be risky because they work at a low level, meaning the programmer has to manage memory manually. This gives more control but also increases the chances of mistakes.

One common problem is called a **buffer overflow**. This happens when you try to store more data than the memory space (the buffer) can hold. When data is written beyond the allocated space, it can damage other data in memory or even create security vulnerabilities that attackers could exploit.

Because C-style arrays and strings do not have built-in checking to stop you from writing too much data, it's the programmer's responsibility to make sure memory is used safely.

Many functions that work with C-style strings, such as `strcpy()` (used to copy strings) and `strcat()` (used to join strings), are not safe if used carelessly. They can cause buffer overflows if the destination array is not large enough to hold the full result.

To reduce this risk, you can use safer versions like `strncpy()` and `strncat()`. These functions let you set a limit on how many characters to copy or join, helping prevent data from going beyond the array's boundaries. Many functions that work with C-style strings can be dangerous if not used carefully. Let's look at some of them:

`strcpy()`

- **Meaning**: Copy one string into another.
- **Risk**: If the destination array is too small, it will cause an error.

- Example of using `strcpy()`:

```
char source[] = "Hello";
char destination[10];
strcpy(destination, source);  // Copies "Hello" safely
```

- If `destination` was smaller than `"Hello"`, this could crash your program.

`strncpy()`

- **Meaning**: Copy part of a string safely by setting a limit.
- You can tell it how many characters to copy.
- Example of using `strncpy()`:

```
char source[] = "Hello";
char destination[4];
strncpy(destination, source, 3);  // Copies only "Hel"
destination[3] = '\0';            // End the string with '\0'
```


- Always add `'\0'` (the null character) at the end yourself.

`strcat()`

- **Meaning**: Add one string to the end of another.
- **Risk**: If there isn't enough space, it can cause an error.
- Example of using `strcat()`:

```
char text[20] = "Good ";
char word[] = "morning";
strcat(text, word);  // Now text becomes "Good morning"
```

- Make sure `text` has enough space for both words.

`strncat()`

- Meaning: A safer version of `strcat()` that lets you set a limit.

- Example of using `strncat()`:

```
char text[20] = "Good ";
char word[] = "morning";
strncat(text, word, 4);  // Adds only "morn"
```


Overall, when using C-style arrays and strings, always be cautious about memory management to prevent errors and protect the integrity of your programs.

Ready to see these concepts in action? Run the demo below to explore the examples and experiment on your own!

Demo 4.1: ArraysStrings/Demos/ArraysAndStrings.cpp

```
1.   #include <iostream>
2.   #include <cstring> // For strcpy(), strncpy(), strcat(), strncat()
3.
4.   int main() {
5.       // 1. Memory Basics: Bits, Bytes, and Words
6.       std::cout << "=== Memory Basics ===\n";
7.       std::cout << "A 'bit' is the smallest unit of data (0 or 1).\n" ;
8.       std::cout << "A 'byte' is 8 bits (enough to store one character).\n" ;
9.      std::cout << "A 'word' is usually 64 bits (8 bytes) on modern computers.\n\n";
10.      std::cout << "=== Example: Size of Data Types ===\n";
11.      std::cout << "Size of char: " << sizeof(char) << " byte\n";
12.      std::cout << "Size of int: " << sizeof(int) << " bytes\n";
13.      std::cout << "Size of double: " << sizeof(double) << " bytes\n\n";
14.
15.      // 2. Working with C-style Arrays
16.      std::cout << "=== C-style Arrays ===\n";
17.      int numbers[5] = {10, 20, 30, 40, 50}; // Array of 5 integers
18.      std::cout << "The array contains: \n";
19.      for (int i = 0; i < 5; i++) {
20.          std::cout << numbers[i] << " \n";
21.      }
22.      std::cout << "Remember: array indexes start at 0 and end at 4.\n";
23.      std::cout <<"Accessing an index outside this range can cause errors.\n\n";
24.
25.      // 3. Exploring C-style Strings
26.      std::cout << "=== C-style Strings ===\n";
27.      char greeting[] = "Hello"; // A string is a char array ending with '\0'
28.      std::cout << "Greeting: " << greeting << "\n";
29.     std::cout << "Each character is stored in memory, ending with a '\\0' sym  ↵↵
           bol.\n\n";
30.
31.      // 4. String Copy Functions
32.      std::cout << "=== strcpy() and strncpy() ===\n";
33.      char source[] = "Good Morning";
34.      char destination1[20];
35.      char destination2[6];
36.
37.      // strcpy() copies the entire string
38.      strcpy(destination1, source);
39.      std::cout << "Using strcpy(): " << destination1 <<"\n";
40.
41.      // strncpy() copies only part of string (safer)
42.      strncpy(destination2, source, 5);
43.      destination2[5] = '\0'; // Always add '\0' manually
```

Demo 4.1: ArraysStrings/Demos/ArraysAndStrings.cpp

```
44.       std::cout << "Using strncpy(): " << destination2 << "\n\n";
45.
46.       // 5. String Concatenation Functions
47.       std::cout << "=== strcat() and strncat() ===\n";
48.       char text[30] = "Good ";
49.       char word[] = "Evening";
50.
51.       // strcat() joins two strings
52.       strcat(text, word);
53.       std::cout << "Using strcat(): " << text << "\n";
54.
55.       // strncat() joins part of a string safely
56.       char text2[30] = "Good ";
57.       strncat(text2, word, 4);
58.       std::cout << "Using strncat(): " << text2 << "\n\n";
59.
60.       // 6. Summary
61.       std::cout << "=== Summary ===\n";
62.       std::cout << "- Arrays store multiple items of the same type.\n";
63.       std::cout << "- C-style strings are character arrays ending with '\\0'.\n";
64.       std::cout << "- strcpy() and strcat() can be unsafe if space is too small.\n";
65.       std::cout << "- strncpy() and strncat() are safer since they limit copied
              data.\n";
66.       std::cout << "- Always check your array size to avoid memory errors.\n\n";
67.       std::cout << "End of Demo. Try changing the text or numbers to experiment!\n";
68.
69.       return 0;
70.   }
```


Running this program on Windows:

1. Open Visual Studio Code.
2. From the **File** Menu select **Open Folder....**
3. Navigate to **ArraysStrings\Demos**. Click **Select Folder**.
4. The folder will appear in the **Explorer** view in the left hand panel. Select **ArraysAndStrings.cpp** so that the file will open in the editor.
5. Run the file by pressing **Ctrl** + **F5**.
6. The output will display in the terminal at the bottom of the screen.

Running this program on Mac:

1. Follow Steps 1 through 4 above for running on Windows.
2. Right click the **Demos** folder in the **Explorer** panel. Select **Open In Integrated Terminal**.
3. You will now see a terminal view beneath the editor. Copy and paste the following command into the terminal and press **Enter**.

```
clang++ -std=c++20 ArraysAndStrings.cpp -o output
```

4. Run the program by typing the following into the terminal and pressing **Enter**:

```
./output
```

Running C++ in the browser

To quickly testing the demos throughout the course, you can use `cpp.sh (https://cpp.sh/)`, a free online platform to run C++ code in your browser. Simply copy and paste your code into the editor and select **Run**. Make sure that you have selected C++20 as your compiler.

While it's not a substitute for a full development environment, it's a convenient option for trying out small C++ programs.

4.2. Modern Arrays: std::array

❖ 4.2.1. Introducing std::array in Modern C++

C++ makes working with arrays safer and easier than the old C-style arrays. With C-style arrays, you had to manually track the size and be careful to avoid memory mistakes.

To solve this, C++ introduced `std::array`, a template-based container (a reusable class that works with any data type) from the **Standard Template Library (STL)**. It provides the same fixed-size structure as regular arrays but with extra safety and helpful functions built in.

Example

```
#include <array>
std::array<int, 3> nums = {1, 2, 3};  // safer and more flexible than C-style arrays
```

- `#include <array>` allows us to use `std::array`.
- `std::array<int, 3>` declares an array of 3 integers.
- `nums` is name of the array.
- `= {1, 2, 3}` initializes the array with values.

Benefits:

- Works like a normal array but **safer** because it knows its size and supports bounds-checked access with `.at()`.
- Comes from the Standard Template Library (STL), so it includes **built-in functions** like `.size()` and works with standard algorithms.
- Size is **fixed** and known at compile time, which makes the code more predictable and efficient.

❖ 4.2.2. Understanding std::array

The `std::array` is a template class (a reusable code pattern) in the C++ Standard Library. A template class means it can work with any data type (`int`, `double`, `string`) while keeping the same structure. It provides a fixed-size array that come with helpful built-in features, making them safer and easier to use than traditional arrays.

Key Features:

Here are the key features of `std::array`:

- **Size Awareness:**

  `std::array` automatically keeps track of its size. This helps prevent out-of-bounds access (trying to access elements that don't exist in the array).

- **Standardized Interface:**

Has built-in functions like `at()`, `size()`, `begin()`, and `end()` for easy access and iteration. These make it easy to get the array size, access elements, and use loops with less manual work.

- **Integration with Standard Algorithms:**

  You can directly use `std::array` with C++ Standard Library algorithms such as `sort()`, `reverse()`, or `find()`, making your code cleaner and more consistent.

- **Safety:**

  The `at()` function performs a safety check before accessing an element. If the index is invalid, it throws an error instead of causing unpredictable behavior.

Example

```
#include <array>
#include <iostream>
int main() {
    std::array<int, 3> nums = {10, 20, 30};
    std::cout << nums.at(1);  // prints 20 safely
}
```


Key Points to Remember:

- Include the `<array>` header to use `std::array`.
- Its size is fixed at compile time.
- `.at()` helps avoid crashes by checking array boundaries.

❖ 4.2.3. std::array Example

The following example shows how to create and use a `std::array`. To understand the improvement, we'll also look at a **C-style array** for comparison.

```
#include <iostream>
#include <array>

int main(){
    // create a C-style array
    std::string fruits_C[] = { "Apples", "Bananas", "Pears", "Oranges" };
    // iterate through the C-style array
    std::cout << "Iterating through C-style array:\n";
    for (int i = 0; i < 4; i++) {
        std::cout << fruits_C[i] << "\n";
    }
    // create a modern C++ array using the same data through std::array
    std::array<std::string, 4> fruits = { "Apples", "Bananas", "Pears", "Oranges" };
    // iterate through the array
    std::cout << "Iterating through modern C++ array:\n";
    for (auto& fruit : fruits) {
        std::cout << fruit << "\n"; }
}
```

Code Breakdown:

- `#include <array>` allows using the `std::array` feature.
- A **C-style array** (`fruits_C`) is created and printed using a normal `for` loop.
- A modern `std::array` is created with the same data.
- The range-based for loop (`for (auto& fruit : fruits)`) makes iteration easier.
- Both loops print the fruits, but `std::array` is safer and more flexible.

❖ 4.2.4. Things to Know About std::array

- `<std::string, 4>`

  Defines a **class template**, meaning it sets the data type (`std::string`) and the number of elements (4) in the array.

- `#include <array>`

  Adds the header file needed to use `std::array` in your program.

- **Array index starts at 0**

The first element has index **0**, and the last has index size **- 1**.

- **Accessing Elements**

```
fruits[1];      // normal access
fruits.at(1);   // safer access
```

`.at()` checks if the index is valid and throws an **out of range** error if it's not.

- **Looping**
  - C-style arrays use the classic for loop.
  - `std::array` supports the simpler range-based for loop:

```
for (auto& fruit : fruits) {
    std::cout << fruit << "\n";
}
```


❖ 4.2.5. Creating an Empty std::array

You can create an empty `std::array` by leaving the curly braces `{}` empty. This means the array will be created with a fixed size but no initial values.

```
#include <array>
std::array<int, 25> myEmptyArray{};  // creates an array of 25 integers
myEmptyArray[0] = 45;                // assign a value to the first element
```

For an array of type `int`, all elements are automatically **initialized to 0** when you leave the braces empty.

❖ 4.2.6. std::array Demo

The following demo program shows how to create, access, and use `std::array` in C++. It also compares it with a traditional C-style array and demonstrates common functions like `.size()`, `.at()`, `.begin()` and `.end()`.

Demo 4.2: ArraysStrings/Demos/ModernArrays.cpp

```cpp
#include <iostream>
#include <array>
#include <algorithm> // for sort()
#include <string>

int main() {
    // Modern Arrays in C++
    std::array<int, 3> nums = {1, 2, 3};
    std::cout << "First element: " << nums[0] << "\n";
    std::cout << "Array size: " << nums.size() << "\n\n";

   std::array<std::string, 4> fruits = {"Apples", "Bananas", "Pears", "Oranges"};
    std::cout << "Second fruit (safe access): " << fruits.at(1) << "\n\n";

    // Example (C-style vs Modern)
    std::string fruits_C[] = {"Apples", "Bananas", "Pears", "Oranges"};

    std::cout << "Iterating through C-style array:\n";
    for (int i = 0; i < 4; i++) {
        std::cout << fruits_C[i] << "\n";
    }

    std::cout << "\nIterating through std::array:\n";
    for (auto& fruit : fruits) {
        std::cout << fruit << "\n";
    }
    std::cout << "\n";

    // Accessing Elements
    std::cout << "Accessing elements:\n";
    std::cout << "Using []: " << fruits[2] << "\n";
    std::cout << "Using .at(): " << fruits.at(2) << "\n\n";

    // Using std::array Functions
    std::array<int, 5> numbers = {5, 2, 9, 1, 3};
    std::sort(numbers.begin(), numbers.end());

    std::cout << "Sorted numbers: ";
    for (int n : numbers) {
        std::cout << n << " ";
    }
    std::cout << "\n\n";

    // Creating an Empty std::array
```

Demo 4.2: ArraysStrings/Demos/ModernArrays.cpp

```
45.       std::array<int, 5> emptyArray{}; // all elements initialized to 0
46.       emptyArray[0] = 10;
47.       std::cout << "First element of empty array: " << emptyArray[0] << "\n";
48.       std::cout << "Default-initialized element: " << emptyArray[1] << "\n\n";
49.
50.       std::cout << "Demo complete.\n";
51.   }
```


4.3. Strings with std::string and std::string_view

Modern C++ provides `std::string` and `std::string_view` for working with text. They make string handling easier, safer, and more efficient than C-style strings.

- `std::string` stores and manages text you can change.
- `std::string_view` lets you access text without creating a new string, which makes it faster and uses less memory.
- Both come from the **STL**, giving you helpful functions like `size()`.

Example

```
std::string name = "Alice";
std::string_view view = name;  // refers to the same text without copying
```

❖ 4.3.1. std::string

`std::string` is a powerful text-handling class in C++. It behaves like a normal string but comes with many built-in features that make working with text easier.

- It **grows** automatically, so you don't worry about fixed size (e.g., `s += " world";`).
- You can **add** strings together using `+` (e.g., `a + b`).
- You can **compare** strings using operators like `==` (e.g., `a == b`).

- You can **access** characters using `[ ]` (e.g., `word[0]`).
- It provides useful functions such as `substr()`, `size()`, and `empty()`.

Example: Using std::string

```
std::string myString = "This is a C++ string";
std::cout << "My String: " << myString << "\n";

// get part of the string
std::cout << "My String.substr(5,2): " << myString.substr(5,2) << "\n";

// access a single character
std::cout << "My String[0]: " << myString[0] << "\n";

// string concatenation
std::string anotherString = myString + " with concatenation!";
std::cout << anotherString << "\n";
```

Code breakdown:

- Creates a string called `myString`.
- Prints the whole string.
- Uses `.substr(5, 2)` to get 2 characters starting at index 5.
- Accesses the first character with `[0]`.
- Adds another piece of text to the original string using `+`.

Output

```
My String: This is a C++ string
My String.substr(5,2): is
My String[0]: T
This is a C++ string with concatenation!
```

Note the following:

- The `substr` member function lets you extract a smaller string from a larger string. You simply give it two numbers:
    - where to start (the index).
    - the length of the desired substring. This makes it easy to get specific words or parts of a sentence just by telling how many characters you want.

- You can access a single character in the string using bracket notation ([ ]). A `std::string` stores its characters in order, so you can treat it like an array of characters. For example, `myString[0]` gives you the first character.
- Use the `+` operator to combine two strings into one. This is called **concatenation**. It allows you to build longer messages or join separate pieces of text easily.

❖ 4.3.2. std::string_view

`std::string_view` gives you a **read-only** view of an existing string. It does not store its own copy of the text. It simply points to text that already exists (a C-style string or a `std::string`).

This makes it very fast and efficient, because:

- It does not copy the characters.
- It only holds a **pointer** to the text and the **size** of the text.
- You can pass large strings around without slowing down your program.

Example: Using std::string_view

```
// use string_view for the substring
std::string myName = "Stephen Withrow";
size_t lastNameIndex = myName.find("Withrow");

std::string_view myLastName(myName.c_str() + lastNameIndex, 7);
std::cout << "My last name using string_view: " << myLastName << "\n";
```


Code breakdown:

- `myName` stores the full name as a normal `std::string`.
- `.find("Withrow")` searches for the starting position of the word “Withrow”.
- `myName.c_str() + lastNameIndex` gives us a pointer to the first character of the last name.
- `std::string_view myLastName(..., 7)` creates a view of 7 characters starting at that position.
- No copying happens. `string_view` just points to part of the original string.

Output

```
My last name using string_view: Withrow
```

Note the following:

- Use the `find` member function of `std::string` to locate the starting index of a substring, like `"Withrow"` in this example.
- `size_t` is an unsigned (meaning, it can't be negative) integer data type used for sizes and indexes. It is defined in the standard C++ header `<cstddef>`.
- `std::string_view` works with C-style strings, so you need to use `c_str()` to convert a `std::string` into a C-style string.
- When creating a `string_view` for a substring, you must pass the length of the substring (7 in this example) along with the pointer to the starting character.

❖ 4.3.3. Key Points to Remember:

For `std::string`:

- Stores text dynamically and grows automatically.
- Can concatenate strings using `+`.
- Supports character access with `[ ]`.
- Use `.substr(start, length)` to get part of a string.
- Comes with many helpful functions like `.size()`, `.empty()`, and `.find()`.
- Safer and easier to use than C-style character arrays.

For `std::string_view`:

- Provides a read-only view of an existing string and no copy is made.
- Efficient for passing around large strings.
- Works with both C-style strings and `std::string`.
- Use `.c_str()` to convert a `std::string` when creating a `string_view`.
- Must provide the length when creating a substring view.

❖ 4.3.4. Demo: Using std::string and std::string_view

The following program demonstrates how to create, manipulate, and view strings in modern C++ using `std::string` and `std::string_view`.

Demo 4.3: ArraysStrings/Demos/StringsWithStdStringView.cpp

```
1.  #include <iostream>
2.  #include <string>
3.  #include <string_view>
4.
5.  int main() {
6.      // std::string Example
7.      std::string myString = "This is a C++ string";
8.      std::cout << "My String: " << myString << "\n";
9.
10.     // substring using substr()
11.     std::cout << "myString.substr(5, 2): " << myString.substr(5, 2) << "\n";
12.
13.     // accessing single character
14.     std::cout << "myString[0]: " << myString[0] << "\n";
15.
16.     // string concatenation
17.     std::string anotherString = myString + " with concatenation!";
18.     std::cout << anotherString << "\n";
19.
20.     // string size and empty check
21.     std::cout << "Size of myString: " << myString.size() << "\n";
22.     std::cout << "Is myString empty? " << (myString.empty() ? "Yes" : "No") <<
            "\n\n";
23.
24.     // std::string_view Example
25.     std::string fullName = "Stephen Withrow";
26.     size_t lastNameIndex = fullName.find("Withrow");
27.
28.     // create a view over the last name
29.     std::string_view myLastName(fullName.c_str() + lastNameIndex, 7);
30.     std::cout << "My last name using string_view: " << myLastName << "\n\n";
31.
32.     // Combining string and string_view
33.     std::string greeting = "Hello";
34.     std::string_view greetView = greeting;
35.     std::cout << "Original greeting: " << greeting << "\n";
36.     std::cout << "View of greeting: " << greetView << "\n\n";
37.
38.     // More examples
39.     std::string hello = "Hello";
40.     std::string world = "World";
41.
42.     // concatenation
43.     std::string message = hello + " " + world;
```

Demo 4.3: ArraysStrings/Demos/StringsWithStdStringView.cpp

```
44.       std::cout << "Concatenated message: " << message << "\n";
45.
46.       // substring using string_view
47.       std::string_view helloView(hello.c_str(), hello.size());
48.       std::cout << "Substring view of hello: " << helloView << "\n";
49.
50.       return 0;
51.   }
```


4.4. String Formatting with std::format

The C++20 standard introduced `std::format`, a modern and convenient way to build formatted strings. Formatting lets you control how text and values appear when printed, making output clearer and more readable.

It is more powerful and flexible than older techniques like string concatenation. With `std::format`, C++ gets Python-style formatting, which makes it easier to build clean, readable strings and control how values appear.

`std::format` lets you write a format string that contains `{}` placeholders. Each `{}` is replaced by a value you provide. This gives you precise control over how numbers, text, and other values appear in the final string.

Key Ideas to Understand:

- You place `{}` wherever you want a value to be inserted in the text.
- You pass the values after the format string, and `std::format` fills them in for you.
- You don't need to manually convert numbers to strings.
- You avoid messy `+` concatenation, making your code cleaner.
- You can also control formatting details like number width, alignment, and decimals (we will see examples later).

- This makes `std::format` a safer and more readable way to create formatted output in modern C++.

Example

```
#include <iostream>
#include <format>

int main() {
    int myInt = 42;
    std::string myCity = "Warner Robins";
    std::string formatted = std::format("My integer: {}, My city: {}", myInt, myCity);
    std::cout << formatted << "\n";
}
```

- `int myInt` declares an integer.
- `std::string myCity` declares a string.
- `std::format("My integer: {}, My city: {}", myInt, myCity);` fills {} placeholders with the given values.
- `std::cout << formatted;` prints the formatted string to the console.
- `#include <format>` supplies the header file for the format function.
- {} marks the spots where values will be inserted.
- The values of the variables are provided after the format string, and each value is inserted into the string at the corresponding {} placeholder.

Output

```
My integer: 42, My city: Warner Robins
```

❖ 4.4.1. Advanced Formatting with std::format

Formatting includes things like spacing, alignment, and how many decimal places a number shows.

With `std::format`, you can do more than just insert values. You can control how they appear using **format specifiers** inside the {} placeholders. The syntax for specifiers is:

```
[fill + align] [sign] [#] [0] [width] [precision] [L] [type]
```

- `[fill + align]`:

    - `fill` is the character used to fill empty space.
    - `align` sets where the value appears: `<` = left, `>` = right, `^` = center.
    - Example: `{:*>10}` right-align in 10 spaces, fill with `*`.

- `[sign]`:
    - `+` always show `+` or `-` for numbers
    - `-` show sign only for negative (default)
    - (space) prefix positive numbers with a space.
    - For example `{:+d}` gives output: `+5`

- `[#]`:
    - Alternate form for numbers.
    - Adds `0x` for hex, `0` for octal, keeps decimal point for floats.
    - Example: `{:#x}` becomes `0x2a`

- `[0]`:
    - Pads numbers with zero instead of spaces.
    - Example: `{:05d}` becomes `00042`

- `[width]`:
    - Minimum number of characters the value should occupy.
    - Example: `{:6}` becomes `"    42"`

- `[precision]`:
    - For floats, controls decimal places.
    - Example: `{:.2f}` becomes `3.14`

- `[L]`:
    - Locale-specific formatting.

- `[type]`:
    - Specifies number or text type:
        - `d` = decimal integer
        - `x` = hexadecimal
        - `b` = binary

- `f` = fixed-point float

Each part is optional, but you can combine them to format your output precisely.

```
#include <format>
#include <iostream>
int main()
{
int itemLength = 25;
std::string itemDescription = "Garden Hose";
float itemPrice = 24.99F;
std::cout << std::format(
    "The item is a {:>4} foot {:<15} that costs ${:.2f}.\n",
    itemLength, itemDescription, itemPrice
);
}
```

Explanation:

- Inside the quotes, the {} are placeholders. Each placeholder has **format specifiers** that control how the output looks:
- `{:>4}`
    - `>` means **right-align** the value inside the space.
    - `4` means the value should take at least 4 characters of space.
    - This makes the number line up nicely with other numbers.
- `{:<15}`
    - `<` means **left-align** the text.
    - `15` means give the text 15 characters of space.
    - This keeps long and short names aligned in columns.
- `{:.2f}`
    - `.2` means show exactly 2 decimal places.
    - `f` means format the number like a normal decimal number (a floating-point format).
    - This is useful for prices or measurements.

The output for this will be:

```
The item is a   25 foot Garden Hose     that costs $24.99.
```

Text Alignment with std::format

You can control text alignment inside placeholders using these symbols:

- `<` for left-align
- `>` for right-align
- `^` for center-align

Width and Fill Character

Width is simply the amount of space you want something to take when it is printed. If your text is shorter than that space, C++ fills the remaining space with a character you choose. This is helpful when you want your output to line up neatly.

For example, `{:*<20}` means:

- Make the text take up 20 characters of space
- Place the text on the left
- Fill the empty space on the right with `*`

Example

```
std::cout << "********************\n";
std::cout << std::format("{:<20}\n", itemDescription); // left
std::cout << std::format("{:>20}\n", itemDescription); // right
std::cout << std::format("{:^20}\n", itemDescription); // center
std::cout << std::format("{:*<20}\n", itemDescription); // left aligned, filled with '*'
```

- The number after the alignment symbol (e.g., `20`) sets the **width** of the field.
- Alignment symbols work only inside the `{}`.
- `{:*<20}` means left-aligned in 20 characters, empty space filled with `*`.
- This makes output neat and easy to read, especially in tables or lists.

The output for this code will be:

```
********************
Garden Hose
         Garden Hose
    Garden Hose
Garden Hose*********
```

Additional Key Specifiers

- **Sign:**
  - `+` always show + or - for numbers
  - `-` show sign only for negative (default)
  - Example:
  - ```
    int n = 5;
    std::cout << std::format("{:+d}\n", n); // Output: +5
    ```

- **Width:**
  - Minimum number of spaces the value should take.
  - Example:
  - ```
    int n = 42;
    std::cout << std::format("{:5d}\n", n); // Output: "   42" (3 spaces + 42)
    ```


- **Precision:**
  - Used for floats, it controls decimal places.
  - Example:
  - ```
    float pi = 3.14159;
    std::cout << std::format("{:.2f}\n", pi); // Output: 3.14
    ```

- **Type:**
  - Tells how the value should appear:
    - `d` for decimal integer
    - `x` for hex
    - `b` for binary
    ```

        - `f` for fixed-point float
    - Example:


```
int n = 42;
std::cout << std::format("{:x}\n", n); // Output: 2a
```

❖ 4.4.2. Demo: Using std::format

The following program demonstrates how to use `std::format` for building formatted strings, including placeholders, multiple values, alignment, width, fill characters, and float precision.

Demo 4.4: ArraysStrings/Demos/UsingStdFormat.cpp

```
1.   #include <iostream>
2.   #include <format>
3.   #include <string>
4.
5.   int main() {
6.       // Basic variables
7.       int myInt = 42;
8.       std::string myCity = "Warner Robins";
9.       std::string itemDescription = "Garden Hose";
10.      int itemLength = 25;
11.      float itemPrice = 24.99F;
12.      float discount = 5.4567F;
13.
14.      // Basic formatting
15.     std::string basic = std::format("My integer: {}, My city: {}", myInt, myCity);
16.      std::cout << basic << "\n\n";
17.
18.      // Advanced formatting
19.      std::cout << std::format(
20.          "The item is a {:>4} foot {:<15} that costs ${:.2f}.\n\n",
21.          itemLength, itemDescription, itemPrice
22.      );
23.
24.      // Alignment examples
25.      std::cout << "______________________\n";
26.      std::cout << std::format("{:<20}\n", itemDescription); // left
27.      std::cout << std::format("{:>20}\n", itemDescription); // right
28.      std::cout << std::format("{:^20}\n\n", itemDescription); // center
29.
30.      // Width and fill character
31.      std::cout << std::format("{:*<20}\n", itemDescription); // left with '*'
32.      std::cout << std::format("{:*>20}\n", itemDescription); // right with '*'
33.     std::cout << std::format("{:*^20}\n\n", itemDescription); // center with '*'
34.
35.      // Additional specifiers examples
36.
37.      int number = 42;
38.      int negative = -42;
39.      float pi = 3.14159;
40.
41.      // Sign specifier
42.      std::cout << std::format("{:+d}, {:+d}\n", number, negative); // always show
             sign
43.
```

Demo 4.4: ArraysStrings/Demos/UsingStdFormat.cpp

```
44.       // Zero-padding
45.       std::cout << std::format("{:05d}\n", number); // pad with zeros to width 5
46.
47.       // Type specifiers
48.       std::cout << std::format("Decimal: {}, Hex: {:x}, Binary: {:b}\n", number,
               number, number);
49.
50.       // Float precision and fixed-point
51.       std::cout << std::format("Pi rounded to 3 decimals: {:.3f}\n\n", pi);
52.
53.       // Float precision
54.       std::cout << std::format("Original price: ${}\n", itemPrice);
55.       std::cout << std::format("Discounted price: ${:.2f}\n\n", itemPrice - dis ↵↵
               count); // 2 decimal places
56.
57.       // Combining text and variables
58.       std::cout << std::format(
59.           "Shopping Info: Item '{}', Length: {} ft, Price: ${:.2f}, Discount:
               ${:.2f}\n\n",
60.           itemDescription, itemLength, itemPrice, discount
61.       );
62.       std::cout << "End of Demo!\n";
63.
64.       return 0;
65.   }
```

Exercise 4: Exploring Arrays and Strings in C++

15 to 30 minutes

❖ E4.1. Requirements

In this exercise, you will extend the existing **Banking Menu** program by adding transaction logs for deposits and withdrawals. These logs will help track all activity that happens during the program.

The program must:

1. Store all deposits in a deposit log.
2. Store all withdrawals in a withdrawal log.
3. Keep track of each entry using arrays of size 100.
4. Display both logs after the user exits the menu.
5. Use `std::format` to center the log headings and right-align the log entries.
6. Stop printing each log when a log entry is `0` (meaning no more stored transactions).

❖ E4.2. Step-by-Step Instructions

1. Open the file **ArraysAndStrings/Exercises/BankingMenu.cpp** in your text editor.
2. **Create** two arrays:
    A. Create a `std::array<int, 100>` named `depositLog` to store deposit amounts.
    B. Create a `std::array<int, 100>` named `withdrawalLog` to store withdrawal amounts.
3. **Set up** index counters:
    A. Make two `size_t` variables: `depositIndex = 0;` and `withdrawalIndex = 0;`
    B. These will keep track of the next empty position in each array.
4. **Modify** the deposit section of your menu:
    A. Whenever the user makes a deposit, add the deposit amount to the next position in `depositLog`.

B. Increase `depositIndex` so the next deposit is stored in the next slot.

5. **Modify** the withdrawal section of your menu:
   A. Whenever the user makes a withdrawal, store the withdrawal amount in the next position of `withdrawalLog`.
   B. Increase `withdrawalIndex` for the next entry.

6. After the user exits the **while loop** (after choosing `'q'` or `'Q'`):
   A. Print the withdrawal log:
      i. Use `std::format("{:^20}", "Log of Withdrawals")` to center the heading in 20 characters.
      ii. Loop through `withdrawalLog`. For each element:
         a. If the value is `0`, stop the loop.
         b. Otherwise, print it right-justified using: `std::format("{:>20}", amount)`.

7. **Print** the deposit log using the same steps:
   A. Center the heading `"Log of Deposits"`.
   B. Right-align every deposit value.
   C. Stop printing when you encounter a `0`.

8. **Compile and run** your updated program to verify that:
   A. Deposits and withdrawals are recorded correctly.
   B. Logs print out centered and aligned properly.
   C. Printing stops at the first unused element.

Example output (your output may differ):

```
Log of Withdrawals
               120
                50
Log of Deposits
               200
                90
```

Solution: ArraysStrings/Solutions/BankingMenu.cpp

```
1.  #include <iostream>
2.  #include <format>
3.  #include <array>
4.  #include <string>
5.  #include <stdexcept>
6.
7.  const double START_BALANCE = 0.0; // Initial account balance
8.  double balance = START_BALANCE;   // User's account balance
9.  std::string userInput;
10.
11. double withdraw(double amount) {
12.     static int nbrOfWithdrawals = 1;
13.     if (amount > balance) {
14.         std::string error_msg = std::format("Withdrawal rejected: amount {:.2f}
              exceeds balance.\n", amount);
15.         throw std::runtime_error(error_msg);
16.     }
17.     else if (amount <= 0) {
18.         std::string error_msg = std::format("Withdrawal rejected: amount {:.2f}
              is less than or equal to zero.\n", amount);
19.         throw std::runtime_error(error_msg);
20.     }
21.     else {
22.         std::cout << std::format("Number of withdrawals is now {}\n",
             nbrOfWithdrawals);
23.         if (nbrOfWithdrawals > 3) {
24.             amount += 3;
25.         }
26.         nbrOfWithdrawals++;
27.         return balance -= amount;
28.     }
29. }
30.
31. double deposit(double amount) {
32.     if (amount <= 0) {
33.         std::string error_msg = std::format("Deposit rejected: amount {:.2f} is
              less than or equal to zero.\n", amount);
34.         throw std::runtime_error(error_msg);
35.     }
36.     else {
37.         return balance += amount;
38.     }
39. }
40.
41. int main() {
```

Solution: ArraysStrings/Solutions/BankingMenu.cpp

```
42.       std::array<double, 100> depositLog{};
43.       int depositIndex = 0;
44.       std::array<double, 100> withdrawalLog{};
45.       int withdrawalIndex = 0;
46.
47.       do {
48.           // Display menu options
49.           std::cout << "\n Deposit (d)\n";
50.           std::cout << " Withdraw (w)\n";
51.           std::cout << " Balance (b)\n";
52.           std::cout << " Quit (q)\n\n";
53.           std::cout << "Enter choice: ";
54.           std::getline(std::cin, userInput);
55.
56.           if (userInput.empty()) continue;  // ignore blank lines
57.
58.           switch (userInput[0]) {
59.               // Handle deposit
60.               case 'd':
61.               case 'D': {
62.                   double amount;
63.                   std::cout << "Enter amount to deposit: ";
64.                   std::getline(std::cin, userInput);
65.                   try {
66.                       amount = std::stod(userInput);
67.                       balance = deposit(amount);
68.                       depositLog[depositIndex++] = amount;
69.                       std::cout << std::format("Deposited ${:.2f}\n", amount);
70.                   }
71.                   catch (std::invalid_argument& ie) {
72.                    std::cout <<  "Deposit amount must be numeric, amount reject  ↵↵
            ed.\n";
73.                   }
74.                   catch (std::runtime_error& re) {
75.                      std::cout << re.what();
76.                   }
77.                   break;
78.              }
79.              // Handle withdrawal
80.              case 'w':
81.              case 'W': {
82.                   double amount;
83.                   std::cout << "Enter withdrawal amount: ";
84.                   std::getline(std::cin, userInput);
```

Solution: ArraysStrings/Solutions/BankingMenu.cpp

```
85.                 try {
86.                     amount = std::stod(userInput);
87.                     balance = withdraw(amount);
88.                     withdrawalLog[withdrawalIndex++] = amount;
89.                     std::cout << std::format("Withdrew ${:.2f}\n", amount);
90.                 }
91.                 catch (std::invalid_argument& ie) {
92.                  std::cout <<  "Withdrawal amount must be numeric, amount re  ↵↵
            jected.\n";
93.                 }
94.                 catch (std::runtime_error& rte) {
95.                     std::cout << rte.what();
96.                 }
97.                 break;
98.             }
99.             // Display balance
100.            case 'b':
101.            case 'B':
102.               std::cout << std::format("Current balance: ${:.2f}\n", balance);
103.                break;
104.            // Handle quit
105.            case 'q':
106.            case 'Q':
107.                std::cout << std::format("Final balance: ${:.2f}\n", balance);
108.                break;
109.            // Handle invalid input
110.            default:
111.              std::cout << "Invalid selection. Please choose a valid option.\n";
112.        }
113.
114.    } while (userInput[0] != 'q' && userInput[0] != 'Q');
115.
116.    // print withdrawal log
117.    std::cout << std::format("{:^20}\n","Log of Withdrawals");
118.    for (auto& withdrawal : withdrawalLog) {
119.        if (withdrawal == 0) {
120.            break;
121.        }
122.       std::cout << std::format("{:>20}\n", std::format("${:.2f}",withdrawal));
123.    }
124.    // print deposit log
125.    std::cout << std::format("{:^20}\n", "Log of Deposits");
126.    for (auto& deposit : depositLog) {
127.        if (deposit == 0) {
```

Solution: ArraysStrings/Solutions/BankingMenu.cpp

```
128.             break;
129.         }
130.         std::cout << std::format("{:>20}\n", std::format("${:.2f}",deposit));
131.     }
132.     return 0;
133. }
134.
135.
```

Conclusion

In this lesson, you learned the fundamentals of C-style arrays and strings, how data lives in memory (bits, bytes, words), and why bounds safety matters. You practiced indexing, saw how the null terminator defines C-style strings, and compared common string functions (`strcpy`/`strncpy`, `strcat`/`strncat`), understanding how length-limited variants help prevent buffer overflows.

You then moved to modern C++ tools: `std::array` for fixed-size containers with `size()`, `at()`, and algorithm support; `std::string` for dynamic text with concatenation, `substr`, `find`, and indexing; and `std::string_view` for efficient, read-only views using `c_str()` and `size_t`. You also formatted output with `std::format` using placeholders, alignment, width, and precision. We guided you to apply these skills by adding transaction logs to the Banking Menu using `std::array` and producing neatly aligned output with `std::format`.

LESSON 5
Memory and Pointers

Topics Covered

☑ Understanding pointers and memory addresses.

☑ Declaring, initializing, and dereferencing pointers.

☑ Address-of and dereference operators.

☑ Pointer arithmetic with arrays and iteration.

☑ Dynamic allocation and deallocation with new and delete.

☑ Understanding smart pointers: unique_ptr, shared_ptr, weak_ptr.

☑ Comparing pointers and references and when to use each.

Introduction

In this lesson, you will build an intuitive and practical understanding of how C++ uses memory and pointers. You will find variable addresses with the address-of operator (`&`), access and modify data using the dereference operator (`*`), declare and initialize typed pointers safely with `nullptr`, and traverse arrays using pointer arithmetic. You will also manage heap memory with `new` and `delete` to avoid leaks and dangling pointers, and compare pointers with references so you know when to use each one.

We will guide you through modern, safer patterns using smart pointers (`unique_ptr`, `shared_ptr`, `weak_ptr`) and RAII so cleanup happens automatically. By the end, you will be able to reason about addresses, walk contiguous memory, allocate and free resources correctly, and apply these skills in focused demos and an exercise that combines arrays and pointers.

5.1. Basics of Pointers and Addresses

❖ 5.1.1. Understanding Memory and Variables

When you create a variable in C++, the computer reserves a specific spot for it in the memory. Every spot in memory has a unique identifier, just like a house on a street has a unique address.

A **memory address** is that unique location identifier where a variable's value is stored.

The Address-Of Operator (&)

To find the memory address of any variable, we use the address-of operator, which is the ampersand (&).

Let's look at a simple example to see the address of a variable:

```
#include <iostream>
#include <string>

int main() {
    std::string myString = "Hello C++";
    std::cout << "The value of myString is: " << myString << "\n";
    std::cout << "The memory address of myString is: " << &myString << "\n";
    return 0;
}
```

Explanation:

- The first line prints the actual data stored in the `myString` variable.
- The second line uses the `&` operator (`&myString`) to retrieve and print the specific memory address where the string `"Hello C++"` is being held.

Output

```
The value of myString is: Hello C++
The memory address of myString is: 00000061E49BFA38
```

Note: The address shown (the long number) will change every time you run the program.

❖ 5.1.2. What is a Pointer?

A pointer is a special variable that stores a memory address of another variable. It points to another variable's location.

The pointer's type (e.g., `int*`) must match the variable it points to (e.g., `int`).

Creating and Initializing a Pointer

To declare a pointer, you use an asterisk (`*`) before the pointer's name.

- **Declaration:** We declare a pointer using `*`.
- **Initialization:** We initialize it by giving it the address of an existing variable using the `&` operator.

Example: Creating and Initializing a Pointer

```
int myInt = 45;

// Declares an integer pointer named myIntPtr
int* myIntPtr;

// Stores the memory address of myInt into the pointer
myIntPtr = &myInt;

std::cout << "Original myInt value: " << myInt << "\n";
std::cout << "Address stored in myIntPtr: " << myIntPtr << "\n";
```

Explanation:

- We first set `myInt` to `45`.
- We then declare an integer pointer `myIntPtr` (using `int*`).
- The key line is `myIntPtr = &myInt;`. This takes the address of `myInt` and saves it as the value of the pointer variable `myIntPtr`.

- The output confirms that `myIntPtr` now holds a memory address.

Output

```
Original myInt value: 45
Address stored in myIntPtr: 0000005345EFF7B4
```

The Dereference Operator (*)

Dereferencing a pointer means accessing the value that is stored at the address the pointer holds. This is how you use the pointer to read or change the original variable. We use the asterisk (*) operator for dereferencing.

Example: Using Dereference Operator

```
// Continuing from the previous example:
// myInt = 45;
// myIntPtr stores the address of myInt

// Use the * operator on the pointer to get the value it points to
std::cout << "Value retrieved using *myIntPtr: " << *myIntPtr << "\n";
// We can also change the value of the original variable using the pointer
*myIntPtr = 99;

std::cout << "myInt value after change via pointer: " << myInt << "\n";
std::cout << "Value retrieved again using *myIntPtr: " << *myIntPtr << "\n";
```


Explanation:

- In the first print statement, `*myIntPtr` follows the stored address and returns the value "45".
- The line `*myIntPtr = 99;` is powerful. It tells the computer to go to the memory address stored in `myIntPtr` and change the value at that spot to "99".
- As a result, the original variable `myInt` is updated to "99".

Output

```
Value retrieved using *myIntPtr: 45
myInt value after change via pointer: 99
Value retrieved again using *myIntPtr: 99
```

Verifying Address Persistence

We can use the address-of operator (`&`) to quickly check if an address stays the same even when the variable's value changes. You can verify if a variable was updated in the same memory location (updated in place) or if the computer had to move it to a completely new spot in memory.

Example

```
std::string myString = "My String Variable";
std::cout << "Current string value: " << myString << "\n";
std::cout << "The address of myString is: " << &myString << "\n";

// change the value:
myString = "Changed value";
std::cout << "The address of myString after change is: " << &myString << "\n";
```

Output

```
Current string value: My String Variable
The address of myString is: 00000061E49BFA38
The address of myString after change is: 00000061E49BFA38
```


Explanation:

- The output shows that the address has not changed.
- This indicates that the `std::string` value was changed in place and did not require a new memory allocation or moving the variable to a different spot. This can be important when you work with large data structures later.

Null Pointers: Pointing to Nothing

Sometimes, you need to declare a pointer but you don't want it to point to a specific variable yet, or it might not be pointing to a valid memory location. In such cases, it is crucial to set the pointer to `nullptr`.

A null pointer is a pointer that points to nothing. Using `nullptr` prevents your program from accessing random or invalid memory, which often leads to crashes or unpredictable behavior.

Example

```
// Declare an integer pointer, initialize it to nullptr
int* safePtr = nullptr;

std::cout << "The value of safePtr is: " << safePtr << "\n";

// Later make it point to a variable
int data = 100;
safePtr = &data;

std::cout << "The value of safePtr after assignment is: " << safePtr << "\n";
std::cout << "The value it points to is: " << *safePtr << "\n";
```

Output

```
The value of safePtr is: 0
The value of safePtr after assignment is: 0000005345EFF7A0
The value it points to is: 100
```

Explanation:

- When `safePtr` is initialized to `nullptr`, its value is typically represented as `0`.
- This is the safe way to show that the pointer is not currently pointing to any valid memory address.
- Later, we assign it the address of the `data` variable using `&data`, and its value changes to a real memory address, allowing us to dereference it safely.

❖ 5.1.3. Key Points to Remember

- Address-Of (`&`)
  - Use the ampersand (`&`) to find the memory address of a variable.
- Pointer Declaration (`*`)
  - Use the asterisk (`*`) to declare a variable as a pointer (e.g., `int *p;`).
- Dereference (`*`)
  - Use the asterisk (`*`) on a pointer variable to access or modify the value stored at the memory address it holds.
- Type Matching

- A pointer must be the same type as the variable it points to (e.g., an `int` pointer for an `int` variable).

- Pointers to Nothing (Null Pointers)
  - It is important to initialize pointers. If a pointer doesn't point to a valid address, it should be set to `nullptr` (e.g., `int* p = nullptr;`). This prevents your program from crashing by trying to access random, invalid memory.

Pointers vs. References

Pointers should not be confused with references (`&` used differently, e.g., `int& ref = myInt;`), which are aliases for existing variables. References cannot be changed to point to a different variable, while pointers can. This topic will be covered later.

❖ 5.1.4. Demo: Using Pointers and Variables

This section provides a complete, working C++ program that demonstrates all the concepts covered: the address-of operator (`&`), pointer declaration and initialization, dereferencing (`*`), and the use of `nullptr`.

You can compile and run this file to see the results yourself.

Demo 5.1: MemoryPointers/Demos/PointersDemo.cpp

```
1.   #include <iostream>
2.   #include <string>
3.
4.   int main()
5.   {
6.
7.       // 1. Address-of operator (&) and Basic Variable
8.       std::string myString = "My C++ String";
9.       std::cout << "--- Address & Value Check ---\n";
10.      std::cout << "Value of myString: " << myString << "\n";
11.      std::cout << "Address of myString (&myString): " << &myString << "\n\n";
             // Use & to get memory address
12.
13.
14.      // 2. Creating, Initializing, and Using a Pointer
15.      int myInt = 45;
16.      int* myIntPtr = &myInt;  // Declares an integer pointer and initialize with
              address of myInt
17.      std::cout << "--- Pointer Initialization ---\n";
18.      std::cout << "Original myInt value: " << myInt << "\n";
19.      std::cout << "Address stored in myIntPtr: " << myIntPtr << "\n\n";  // my  ↵↵
             IntPtr holds the address of myInt
20.
21.
22.      // 3. Dereferencing Pointers (*)
23.      std::cout << "--- Dereferencing ---\n";
24.      // Read the value using the dereference operator (*)
25.      std::cout << "Value using *myIntPtr (Dereferenced): " << *myIntPtr << "\n";

26.      // Change the value of the original variable (myInt) using the pointer
27.      *myIntPtr = 99;
28.      std::cout << "myInt value after being changed via pointer: " << myInt <<
             "\n";
29.      std::cout << "Value using *myIntPtr after changing: " << *myIntPtr << "\n\n";

30.
31.
32.      // 4. Address Persistence Check
33.      std::string checkString = "Verify Address";
34.      std::cout << "--- Address Persistence ---\n";
35.      // Get the initial address
36.      std::cout << "Original address: " << &checkString << "\n";
37.      // Change the value
38.      checkString = "Address Stays Same";
39.      // Check the address again; it should be the same
```

Demo 5.1: MemoryPointers/Demos/PointersDemo.cpp

```
40.       std::cout << "Address after change: " << &checkString << "\n\n";
41.
42.
43.       // 5. Null Pointer Example
44.       std::cout << "--- Null Pointer Example ---\n";
45.       // Initialize a pointer to nullptr
46.       int* safePtr = nullptr;
47.       std::cout << "The value of nullptr is: " << safePtr << "\n";
48.
49.       // Assign a valid address
50.       int data = 100;
51.       safePtr = &data;
52.      std::cout << "The value of nullptr after assignment is: " << safePtr << "\n";
53.       std::cout << "The value it points to is: " << *safePtr << "\n\n";
54.       std::cout<<"End of Demo!";
55.       return 0;
56.   }
```


5.2. Pointer Arithmetic

❖ 5.2.1. Pointer Arithmetic and Arrays

Pointer arithmetic allows you to perform addition or subtraction directly on a pointer's memory address. This is a crucial concept because it lets you manually move a pointer to the next or previous item in a sequence of memory, such as an array.

When you add 1 to an `int` pointer, C++ automatically increments the address by the size of an `int` (typically 4 bytes), ensuring the pointer moves to the very next `int` in memory.

Advancing a Pointer in an Array

When working with arrays, the elements are stored in **contiguous memory** (right next to each other). Pointer arithmetic takes advantage of this.

If an integer array stores 5 integers, and you increment the pointer by 1, it automatically skips exactly one 4-byte chunk (one `int`) and points to the next integer.

Let's look at manually advancing a pointer through an array of integers.

```
#include <iostream>
#include <array>

int main() {
    std::array<int, 5> myInts = { 1, 2, 3, 4, 5 };
    // Get the address of the first element using .data()
    int* arrayPtr = myInts.data();

    std::cout << "First item in the array: " << *arrayPtr << "\n";
    std::cout << "Second item in the array: " << *++arrayPtr << "\n";

    return 0;
}
```


Note the following:

- `arrayPtr` is initialized to the memory location of the first element (which has the value 1) using `myInts.data()`.
- The first `std::cout` line prints the value at the current address: `*arrayPtr` gives us 1.
- In the second print line, the expression `*++arrayPtr` first increments the pointer (`++arrayPtr`) so it now points to the next integer in memory (which has the value 2). Then, the `*` operator dereferences it to print the value 2.

Here is the output:

Output

```
First item in the array: 1
Second item in the array: 2
```

Iterating through an Array

We can use pointer arithmetic inside a `for` loop to manually visit every element in the array:

```
#include <iostream>
#include <array>

int main() {
    std::array<int, 5> myInts = { 1, 2, 3, 4, 5 };
    std::cout << "Iterate through the array using pointer arithmetic: \n";
    // Set arrayPtr to start at the beginning of the array
   for (int* arrayPtr = myInts.data(); arrayPtr < myInts.data() + myInts.size(); ++ar ↵↵
rayPtr)
    {  std::cout << *arrayPtr << " ";  }
    // Loop continues as long as arrayPtr is before the address of the element after the
 end
    std::cout << "\n";
    return 0;
}
```

Items of interest:

- `int* arrayPtr = myInts.data();` sets the pointer to the **starting address** of the array.
- `arrayPtr < myInts.data() + myInts.size();` is the **loop control condition**. It compares the current address (`arrayPtr`) to the address just past the end of the array. The loop runs as long as the **condition is true**.
    - `myInts.data()`: Returns the starting memory address.
    - `myInts.size()`: Returns the number of items (5).
- `++arrayPtr` moves the pointer to the next integer element in memory for each loop iteration.
- `std::cout << *arrayPtr << " ";` **dereferences** the pointer to print the value at the current location.

And here is the output:

Output

```
Iterate through the array using pointer arithmetic:
1 2 3 4 5
```

Essential Functions

- `myInts.data()`: Returns the memory address of the first element in the array.

- `myInts.size()`: Returns how many items are in the array.

❖ 5.2.2. Cautions with Pointer Arithmetic

While powerful, pointer arithmetic must be handled carefully to maintain program stability and prevent security issues.

- **Boundary Errors:** The most common error is accessing memory out of bounds. This happens when you increment or decrement a pointer beyond the memory block allocated for the data structure (like an array). This can lead to unpredictable behavior or program crashes.
- **Valid Memory:** Always ensure the pointer points to valid memory by staying strictly within the allocated bounds of the array or data structure.
- **Avoid Excessive Arithmetic:** Avoid performing complex arithmetic that moves the pointer far beyond the intended data. Stick to simple increments (`++`) and decrements (`--`) when navigating structures.

Key Points to Remember

- A pointer stores a **memory address**, not a value.
- `ptr + 1` moves the pointer to the **next element**, not the next byte.
- Arrays store elements next to each other in memory, so pointers can walk through them.
- `array.data()` gives the address of the first element.
- `array.size()` tells you how many elements are in the array.
- Always stay within array bounds to avoid crashes.
- Dereference (`*ptr`) to get the value the pointer points to.

❖ 5.2.3. Demo: Using Pointer Arithmetic

This program demonstrates how to manually advance a pointer to access sequential elements in an array and how to iterate through an entire array using pointer arithmetic in a `for` loop.

Demo 5.2: MemoryPointers/Demos/PointerArithmeticDemo.cpp

```cpp
#include <iostream>
#include <array>

int main() {

    std::array<int, 5> nums = {1, 2, 3, 4, 5};

    // Start pointer at first element
    int* ptr = nums.data();
    std::cout << "First value: " << *ptr << "\n";

    // (pointer arithmetic)
    ++ptr;   // Move pointer to next element
    std::cout << "Second value (after ++ptr): " << *ptr << "\n";

    // Move two steps ahead
    ptr += 2;
    std::cout << "Fourth value (after ptr += 2): " << *ptr << "\n";

    // Move pointer one step back
    ptr--;
    std::cout << "Third value (after ptr--): " << *ptr << "\n\n";


    // Loop using pointer arithmetic
    std::cout << "Iterating with pointer arithmetic:\n";
    for (ptr = nums.data(); ptr < nums.data() + nums.size(); ++ptr) {
        std::cout << *ptr << " ";
    }
    std::cout << "\n\n";


    // Demonstrating bounds
    std::cout << "Valid positions:\n";
    for (size_t i = 0; i < nums.size(); ++i)
    {
        std::cout << "*(nums.data() + " << i << ") = " << *(nums.data() + i) <<
          "\n";
    }
    std::cout << "\nAvoid doing this (out of bounds): *(nums.data() + 10)\n";
    std::cout << " this will generate undefined behavior \n\n";
    std::cout << "End of Demo!";

    return 0;
```

Demo 5.2: MemoryPointers/Demos/PointerArithmeticDemo.cpp

```
44.  }
```


5.3. Dynamic Memory: new and delete

❖ 5.3.1. Dynamic Memory Allocation

Up until now, the variables you created were handled automatically by the system (this is called **Stack memory**).

Dynamic memory allocation is a technique that lets you manually control memory. This memory is allocated while the program is running (**runtime**) and is taken from a special area called the **heap**. This gives you flexibility, especially when dealing with data structures whose size isn't known until the program runs.

Core Memory Terms

- **Heap:** A large area of memory from which a program can request (allocate) and return (deallocate) blocks of memory while it is running. This memory lives longer than regular function variables.
- **Runtime:** The period during which a program is executing or running.
- **Stack Memory:** A reserved area of memory that stores variables created inside functions (local variables). This memory is managed automatically by the system. When a function finishes, all its variables stored on the stack are automatically destroyed.

❖ 5.3.2. Dynamic Memory Allocation with new

To request memory from the heap, we use the new operator. The new operator performs two actions:

1. It **finds** a block of memory on the heap large enough for the requested data type.
2. It **returns** the address of that new memory block. This address must be stored in a pointer.

Example: Allocating a `std::string`

We will dynamically allocate memory for a `std::string` variable using `new` :

```
#include <iostream>
#include <string>

int main() {
    // The new operator allocates memory for a string on the heap
    std::string* myName = new std::string("Stephen Withrow");

    // Using dereference operator to access the value stored at the address
    std::cout << "My name allocated from dynamic memory: " << *myName << "\n";
    return 0;
}
```

What's happening here?

- `std::string* myName` means "`myName` is a pointer to a `std::string`".
- `new std::string("Stephen Withrow")` requests memory from the heap and constructs a `std::string` in that memory.
- `*myName` dereferences the pointer to access the actual string value.

Output

```
My name allocated from dynamic memory: Stephen Withrow
```

❖ 5.3.3. Dynamic Memory Deallocation with `delete`

Dynamic memory is not automatically cleaned up. If you allocate memory using `new`, you are responsible for freeing it up using the `delete` operator when you are finished with it.

Memory Leaks:

This occurs when memory is allocated from the heap using `new` but is never released back to the system using `delete`. Over time, a large number of memory leaks can cause a program or system to run out of available memory.

Example: Allocating and Deallocating

```
#include <iostream>
#include <string>

int main() {
    std::string* myName = new std::string("Stephen Withrow");
    std::cout << "Value before delete: " << *myName << "\n";

    // Use delete to return allocated memory back to heap
    delete myName;

    // Set the pointer to nullptr after deletion
    myName = nullptr;
    return 0;
}
```

Explanation:

- After using the memory, `delete myNameDynamic;` frees the memory block that the pointer was pointing to.
- Crucially, the `delete` operator only frees the memory block and does not change the pointer itself. The pointer is then set to `nullptr` for safety.

Note: We cannot safely access `*myName` after deletion.

Output

```
Value before delete: Stephen Withrow
```

❖ 5.3.4. Why Use Dynamic Memory?

Dynamic memory is helpful when:

- You don't know how much memory you need at compile time.
- You want something to stay alive even after the function ends.
- You're creating data structures (linked lists, trees, etc.).

❖ 5.3.5. Safe Practices in Manual Memory Management

Managing memory manually is powerful, but it comes with risks.

- Ensure every `new` has a corresponding `delete` to prevent **memory leaks**. This is the golden rule of manual memory management.
- Set pointers to `nullptr` after deleting their allocated memory.
    - Failing to do this leaves you with a **dangling pointer** (a pointer that holds an address that is no longer valid or owned by your program).
- Use `new[]` to allocate an array dynamically (e.g., `int* arr = new int[10];`). You must use `delete[]` (e.g., `delete[] arr;`) to free it.
- Regularly check for memory allocation failures, particularly in environments with limited resources, by verifying if the returned pointer from `new` is `nullptr`.

❖ 5.3.6. Demo: Dynamic Allocation and Deallocation

Demo 5.3: MemoryPointers/Demos/DynamicMemoryDemo.cpp

```
1.   #include <iostream>
2.   #include <string>
3.
4.   int main() {
5.       // Dynamically allocate a string
6.       std::string* greeting = new std::string("Hello from the heap!");
7.       std::cout << *greeting << "\n";
8.
9.       // Dynamically allocate an int
10.      int* score = new int(100);
11.      std::cout << "Score: " << *score << "\n";
12.
13.      delete greeting; // free memory
14.      greeting = nullptr;
15.
16.      delete score;
17.      score = nullptr;
18.
19.      return 0;
20.  }
```


5.4. Smart Pointers: unique_ptr, shared_ptr, weak_ptr

❖ 5.4.1. Smart Pointers: Safer Memory Management

In modern C++, smart pointers are tools that automatically manage memory allocated on the heap, solving the problems of forgetting to use `delete` (memory leaks) or using `delete` incorrectly (dangling pointers). They behave like regular pointers but have extra logic that ensures cleanup happens automatically.

To use smart pointers be sure to include the `<memory>` header.

❖ 5.4.2. `unique_ptr`: Exclusive Ownership

The `unique_ptr` is the simplest and most common smart pointer. It guarantees exclusive ownership of the memory it manages. This means:

- Only one `unique_ptr` can point to a specific object on the heap at any time.
- When the `unique_ptr` variable goes out of scope (e.g., when a function ends), the memory it points to is automatically and immediately deleted.
- You cannot copy a `unique_ptr` (ownership cannot be shared).

This makes `unique_ptr` ideal for standard heap allocations where you know exactly which part of the code is responsible for the object's lifetime.

Example

```
#include <iostream>
#include <memory>
#include <string>

int main() {
    // Allocates a string dynamically. uniquePtr now owns this memory.
    std::unique_ptr<std::string> uniquePtr = std::make_unique<std::string>("Exclusive
Data");
 std::cout << "Value: " << *uniquePtr << "\n";

    // When main() ends, uniquePtr goes out of scope and delete is called automatically.

    return 0;
}
```

Explanation:

- The `std::make_unique` function is the preferred way to create a `unique_ptr`.
- When the program finishes, the `uniquePtr` is destroyed, and the string memory it managed is automatically freed.
- If you try to copy a `unique_ptr`, C++ would give you an error, enforcing its exclusive ownership rule.
- This is ideal when you want **one clear owner**.

Output

```
Value: Exclusive Data
```


❖ 5.4.3. `shared_ptr`: Shared Ownership

The `shared_ptr` is designed for situations where multiple parts of your program need to **share ownership** of the same object. It keeps track of how many pointers are currently pointing to the object using an internal **reference count**.

- When you copy a `shared_ptr`, the reference count increases.
- When a `shared_ptr` goes out of scope, the reference count decreases.
- The managed object is only deleted when the reference count drops to zero.
- Useful for shared resources.

Example

```
#include <iostream>
#include <memory>

int main() {
    // Creates a shared_ptr. Reference count (RC) = 1.
    std::shared_ptr<int> sharedPtr1 = std::make_shared<int>(150);

    std::cout << "RC before copy: " << sharedPtr1.use_count() << "\n";
 // use_count() returns the RC

    // sharedPtr2 now shares ownership. RC = 2.
    std::shared_ptr<int> sharedPtr2 = sharedPtr1;
    std::cout << "RC after copy: " << sharedPtr1.use_count() << "\n";

    // sharedPtr2 goes out of scope here (if this were inside a function) or when main
ends.
    // The memory is NOT deleted because sharedPtr1 still exists (RC is not zero).
    return 0;
}
```


Explanation:

- The `std::make_shared` function is the best way to create a `shared_ptr`.
- When `sharedPtr2` is created from `sharedPtr1`, the reference count goes to 2.
- The memory containing the value `150` will only be deleted when both `sharedPtr1` and `sharedPtr2` are destroyed.

Output

```
RC before copy: 1
RC after copy: 2
```

❖ 5.4.4. `weak_ptr`: Non-Owning Reference

The `weak_ptr` provides a special, non-owning and a **weak reference** to an object managed by a `shared_ptr`.

- It does not increase the reference count of the `shared_ptr` it observes.
- You cannot directly use it without converting to `shared_ptr`.

- Used to avoid **circular references** (`shared_ptr` loops).
- It is used to safely check if a shared object still exists without keeping it alive indefinitely.

The most common use for `weak_ptr` is to break circular references that can occur when two `shared_ptr` objects point to each other, which would otherwise prevent the memory from ever being freed.

Accessing Data

Because `weak_ptr` does not own the memory, you cannot directly dereference it. You must first convert it into a `shared_ptr` using the `lock()` method to check if the object is still alive.

Example

```
#include <iostream>
#include <memory>

int main() {
    // 1. Create a shared_ptr (RC = 1)
    std::shared_ptr<int> sharedPtr = std::make_shared<int>(404);

    // 2. Create a weak_ptr from the shared_ptr. RC remains 1.
    std::weak_ptr<int> weakPtr = sharedPtr;

    // 3. Lock the weak_ptr to try and get temporary ownership (a new shared_ptr)
    std::shared_ptr<int> tempPtr = weakPtr.lock();

    if (tempPtr) {
        std::cout << "Object exists! Value: " << *tempPtr << "\n";
    } else {
        std::cout << "Object was already destroyed.\n";
    }
    return 0;
}
```


Explanation:

- `weakPtr` observes the memory managed by `sharedPtr`.
- The `lock()` method attempts to upgrade the weak reference to a strong `shared_ptr` (`tempPtr`).

- If `sharedPtr` had already been destroyed (RC was 0), `lock()` would return `nullptr`, allowing us to safely check for the object's existence.

Output

```
Object exists! Value: 404
```

Note: You will learn more about smart pointers in the data structures lesson later in the course.

❖ 5.4.5. Key Points to Remember

- **Safety First:** Smart pointers eliminate most manual memory errors by using RAII (automatic cleanup).
- `unique_ptr`: For **exclusive ownership** (no sharing). It's the default choice for heap allocation.
- `shared_ptr`: For **shared ownership**, tracks the number of owners using a reference count. Memory is deleted only when the count reaches zero.
- `weak_ptr`: For **non-owning access**, often used to break circular dependencies in complex data structures. It requires `lock()` to safely access the managed object.
- `make_unique` / `make_shared`: Always use `std::make_unique` and `std::make_shared` to create smart pointers, as this is safer and more efficient than using `new` directly.

❖ 5.4.6. Demo: Using Smart Pointers

This program demonstrates the core functionality of `unique_ptr`, `shared_ptr`, and `weak_ptr`.

Demo 5.4: MemoryPointers/Demos/SmartPointersDemo.cpp

```
1.  #include <iostream>
2.  #include <memory>
3.  #include <string>
4.
5.  int main() {
6.      // 1. unique_ptr (Exclusive Ownership)
7.      std::cout << "--- unique_ptr Demo ---\n";
8.      // Allocation: Memory is automatically deleted when uniquePtr goes out of
            scope.
9.      std::unique_ptr<std::string> uniquePtr = std::make_unique<std::string>("Unique
            String Data");
10.     std::cout << "unique_ptr value: " << *uniquePtr << "\n\n";
11.
12.     // 2. shared_ptr (Shared Ownership)
13.     std::cout << "--- shared_ptr Demo ---\n";
14.     // RC = 1
15.     std::shared_ptr<int> sharedPtr1 = std::make_shared<int>(150);
16.     std::cout << "sharedPtr1 RC: " << sharedPtr1.use_count() << "\n";
17.
18.     // RC = 2 (Both pointers own the memory)
19.     std::shared_ptr<int> sharedPtr2 = sharedPtr1;
20.     std::cout << "sharedPtr2 created. RC is now: " << sharedPtr1.use_count() <<
            "\n\n";
21.
22.     // 3. weak_ptr (Non-Owning Reference)
23.     std::cout << "--- weak_ptr Demo ---\n";
24.     // weakPtr observes sharedPtr1, but RC remains 2.
25.     std::weak_ptr<int> weakPtr = sharedPtr1;
26.
27.     // Temporarily convert weakPtr to a shared_ptr to access the value
28.     if (auto tempPtr = weakPtr.lock()) {
29.        std::cout << "weak_ptr successfully locked. Value: " << *tempPtr << "\n";
30.     } else {
31.         std::cout << "weak_ptr target object was destroyed.\n";
32.     }
33.     std::cout << "\nEnd of Demo!";
34.     // When main ends, sharedPtr1 and sharedPtr2 are destroyed, RC drops to 0,
            and memory is freed.
35.
36.     return 0;
37. }
```

5.5. Pointers vs. References

Pointers and references are both ways to work with a variable's memory location.

A **pointer** stores the actual memory address of another variable. It can be changed to point to another variable. A **reference** is another name, or alias, for an existing variable. It **cannot be changed** to refer to something else.

❖ 5.5.1. Pointers

A pointer is a variable whose value is a memory address.

- **Syntax:** Pointers are created using the asterisk (*) symbol.
- **Flexibility:** Pointers can be declared and can be easily reassigned to point to different memory locations.

Example: Creating a Pointer

```
#include <iostream>
#include <string>

int main() {
    // A regular variable on the stack
    int myValue = 100;

    // Define myPointer as a pointer (*) to an integer, storing myValue's address (&)
    int* myPointer = &myValue;

    // We must use the dereference operator (*) to access the value
    std::cout << "Value via pointer: " << *myPointer << "\n";

    // Changing the pointer's value
    *myPointer = 200;
    std::cout << "Original value after change using pointer: " << myValue << "\n";

    return 0;
}
```


Items of interest:

- The pointer `myPointer` stores the address of `myValue`.
- To read or modify `myValue` through the pointer, you must explicitly use the **dereference** operator (`*myPointer`).

Output

```
Value via pointer: 100
Original value after change using pointer: 200
```

❖ 5.5.2. References

A reference must be initialized when declared and binds directly to an existing object. It cannot be null and has no separate memory address of its own.

- **Syntax:** References are created using the ampersand (`&`) symbol, placed after the type.
- **Initialization Rule:** A reference must be initialized when it is declared, ensuring it always points to a valid object.

Example: Creating a Reference

```
#include <iostream>
#include <string>

int main() {
    std::string originalName = "Stephen Withrow";
    // Define myNameAlias as a reference (&) to originalName
    std::string& myNameAlias = originalName;

    // Both variables access the same memory location
    std::cout << "Original name: " << originalName << "\n";
    std::cout << "Alias name: " << myNameAlias << "\n";

    // Changing the alias changes the original variable
    myNameAlias = "Steve W.";
    std::cout << "Original name after change using alias: " << originalName << "\n";

    return 0;
}
```


Items of interest:

- `myNameAlias` is another name, or alias, for `originalName`.
- When you change `myNameAlias`, you are directly modifying the data stored in `originalName`'s memory slot.

Output

```
Original name: Stephen Withrow
Alias name: Stephen Withrow
Original name after change using alias: Steve W.
```

❖ 5.5.3. Differences between Pointers and References

- ### Initialization and Assignment
    - **Pointers** (`*`) can be declared without immediate initialization, or they can be initialized to `nullptr` (a pointer that points to nothing). They can also be reassigned during runtime to point to a different object or memory location.

    - **References** (`&`) must be initialized immediately when they are declared. Once a reference is initialized to a variable, it cannot be changed or **reassigned** to refer to another variable for its entire lifetime.

- Dereferencing
    - **Pointers** require explicit dereferencing using the asterisk (`*`) operator to access the value stored at the memory address they hold (e.g., `std::cout << *myPointer;`).
    - **References** do not require explicit dereferencing. They act as an alias, so accessing the value is direct and simple, just like accessing the original variable (e.g., `std::cout << myReference;`).
- Memory Management
    - **Pointers** can be used to manually manage memory on the **heap** using the `new` and `delete` operators, which is essential for creating dynamic data structures.
    - **References** cannot be used for dynamic memory allocation. They only work with variables that already exist in memory (either on the stack or heap) and cannot be null.
- Safety and Syntax
    - **Pointers** are generally considered less safe because they can be uninitialized or set to invalid memory addresses, leading to dangerous behaviors like null pointer exceptions or dangling pointers.
    - **References** are considered safer and more intuitive because they must always be initialized to a valid object, eliminating the risk of being null or uninitialized.

❖ 5.5.4. Key Features of Pointer (*)

Pointers are variables that hold a memory address. They offer maximum control over memory.

- **Memory Management:** Pointers are essential for dynamic memory allocation using the `new` and `delete` operators, allowing you to manually control memory on the heap.
- **Flexibility (Reassignment):** A pointer can be reassigned during runtime to point to different memory locations or variables.
- **Null State:** Pointers can be initialized to `nullptr`, safely indicating that they are not pointing to any valid memory address.

- **Dereferencing:** Requires the explicit dereference operator (`*`) to access or modify the value stored at the address they hold (e.g., `*myPointer = 5;`).
- **Arithmetic:** Pointers support pointer arithmetic, allowing you to manually move to the next or previous element in a sequence (like an array).

❖ 5.5.5. Key Features of References (&)

References are aliases for existing variables. They provide a safer, syntactically simpler way to work with a variable's memory.

- **Alias (Identity):** A reference is an alias for an existing variable. It is not a separate object that holds an address.
- **Fixed Target:** Once initialized, a reference cannot be reassigned to refer to a different variable. It is permanently bound to its initial target.
- **Safety (Non-Null):** References must be initialized immediately upon declaration and cannot be set to `nullptr`, eliminating the risk of null pointer exceptions.
- **Direct Access:** References do not require explicit dereferencing (no `*` needed); they are used directly, just like the original variable.
- **Function Arguments:** They are the preferred and safest way to implement Call by Reference in function arguments, allowing the function to modify the original variable efficiently.

❖ 5.5.6. Demo: Pointers vs. References

This demo illustrates the key features of both pointers and references in a single program.

Demo 5.5: MemoryPointers/Demos/pointerVsRefDemo.cpp

```cpp
#include <iostream>

int main() {
    int score = 10;
    int bonus = 5;

    // REFERENCE
    int& r = score;    // Must be initialized
    int* p = nullptr;  // Pointers can be initialized to null

    int* heapPtr = new int(100);
    std::cout << "--- Initial Values ---\n";
    std::cout << "Score: " << score << "\n";
    std::cout << "Ref (r): " << r << "\n";
    std::cout << "Ptr (*p): Ptr is null\n"; // Cannot safely dereference p yet
    std::cout << "Heap Ptr (*heapPtr): " << *heapPtr << "\n\n";

    // 1. ACCESS & MODIFICATION

    r = 15;  // Reference
    p = &bonus;
    *p = 50;   // Pointer (Requires *)
    std::cout << "--- Access & Modification ---\n";
    std::cout << "Score (via ref): " << score << "\n";
    std::cout << "Bonus (via ptr): " << bonus << "\n\n";

    // --- 2. REASSIGNMENT ---
    int anotherScore = 200;

    // Pointers can change targets
    p = &anotherScore;
    std::cout << "Ptr reassigned to anotherScore: " << *p << "\n";

    // References cannot change targets
    r = anotherScore;
    std::cout << "Score after 'reassignment' attempt: " << score << "\n\n";

    // Cleanup
    delete heapPtr;
    heapPtr = nullptr;

    std::cout<<"End of Demo!";
    return 0;
}
```

Exercise 5: Explore Pointers and Memory

10 to 20 minutes

In this exercise, you will work with two arrays and use pointers to read and modify their values.

❖ E5.1. Requirements

1. Create two arrays of type `int`.
2. Use pointers to iterate through the arrays.
3. Modify the values in one of the arrays using data from the other.

❖ E5.2. Step-by-step Instructions

1. Create a C++ file named `ArraysUsingPointers.cpp` in the Exercises folder of this lesson.
2. Define the following two arrays:

```
std::array<int, 5> myInts10 = { 10, 20, 30, 40, 50 };
std::array<int, 5> myInts100 = { 100, 200, 300, 400, 500 };
```

3. Use a pointer to iterate through `myInts10`. For each element:
    A. Add the corresponding element from `myInts100`.
    B. Assign this sum back to the current `myInts10` element.
    C. You should also use a pointer when accessing the values in `myInts100`.
4. During each iteration, display:
    A. The current value from `myInts10`
    B. The current value from `myInts100`

C. The new updated value stored back into `myInts10`

Expected Output (first two iterations)

```
myInts10 current value: 10
myInts100 value: 100
myInts10 new value: 110
myInts10 current value: 20
myInts100 value: 200
myInts10 new value: 220
```

Solution: MemoryPointers/Solutions/ArraysUsingPointers.cpp

```
1.  #include <iostream>
2.  #include <array>
3.
4.  int main() {
5.      std::array<int, 5> myInts10 = { 10, 20, 30, 40, 50 };
6.      std::array<int, 5> myInts100 = { 100, 200, 300, 400, 500 };
7.
8.      // Get raw pointers to the start of each array
9.      int* arrayPtr10 = myInts10.data();
10.     int* arrayPtr100 = myInts100.data();
11.
12.     // Loop through both arrays using pointer arithmetic
13.     for (size_t i = 0; i < myInts10.size(); ++i) {
14.
15.         // Print current values
16.         std::cout << "myInts10 current value: " << *(arrayPtr10 + i) << "\n";
17.         std::cout << "myInts100 value: " << *(arrayPtr100 + i) << "\n";
18.
19.         // Modify myInts10 by adding the corresponding myInts100 element
20.         *(arrayPtr10 + i) += *(arrayPtr100 + i);
21.
22.         // Print the updated value
23.         std::cout << "myInts10 new value: " << *(arrayPtr10 + i) << "\n";
24.     }
25.
26.     return 0;
27. }
```

Conclusion

In this lesson, we explored the core concepts of memory and pointers for understanding how C++ programs store, access, and manage data. We began by examining how program memory is organized and why proper allocation and deallocation are essential for correct and efficient execution.

We then introduced pointers as variables that store memory addresses and demonstrated how they allow direct access to underlying data. Through practical examples, you saw how pointer arithmetic enables moving through contiguous memory and how dereferencing provides a way to read and modify values at specific addresses.

We also emphasized the importance of safe pointer practices, including avoiding **dangling pointers**, preventing **memory leaks**, and ensuring that dynamically allocated memory is properly released. These habits are essential for building stable, predictable, and high-performance applications.

Overall, memory and pointers offer powerful control over data and program behavior. By developing a solid understanding of these mechanisms, you are now better equipped to write efficient, reliable C++ programs and to approach more advanced topics such as smart pointers, dynamic data structures, and resource management.

LESSON 6
Structured Programming

Topics Covered

☑ Creating user-defined types with struct.

☑ Simplifying type names with typedef.

☑ Passing structs to functions by value and by pointer.

Introduction

In this lesson, you will apply structured programming to design clear, modular C++ programs using user-defined types. You will learn to group related data with `struct`, create readable type aliases with `typedef` (and `using`), and define `enum` values to replace magic numbers with meaningful names. We will explain how to create, initialize, and access struct members, and how to pass structs to functions efficiently by value, reference, const reference, and pointer.

You will practice these skills in a small Banking Menu app: you will model account data in a `struct`, embed an `enum class` for account type, collect input, track transactions, and produce a formatted account report. We will guide you to write code that is easier to understand, maintain, and extend.

6.1. User-Defined Types: struct

In C++, a *user-defined type* is a type that you create yourself to represent something meaningful in your program, often something that cannot be described well using only built-in types like `int`, `double`, or `char`. One of the simplest and most commonly used user-defined types is the `struct` (short

for *structure*). A `struct` allows you to group related pieces of data into a single, organized unit. This makes your code easier to understand, maintain, and expand.

❖ 6.1.1. What is a `struct`?

A `struct` is a container that holds multiple variables, called **members**. These members can be of different types. For example, you might want to represent a *student* with a name, age, and GPA. These values belong together logically, and bundling them into a `struct` keeps them neatly packaged:

```
struct Student {           // Define a new struct type named "Student"
    std::string name;      // Member to store the student's name
    int age;               // Member to store age
    double gpa;            // Member to store GPA
};
```

Here, `Student` becomes a new type that you can use just like the built-in ones.

❖ 6.1.2. Why Use `struct`?

Using `struct` helps you:

- Organize data logically - All related information stays together.
- Avoid confusion - Instead of keeping separate variables, like `studentName`, `studentAge`, and `studentGpa`, you can keep them inside one object.
- Write clearer code - Functions can accept a single `struct` instead of many separate arguments.
- Build more complex programs - `struct` is the foundation for more advanced features such as classes and object-oriented programming.

❖ 6.1.3. Default Member Initialization

C++ allows you to give members default values directly in the `struct`:

```
struct Point {
    double x = 0.0;    // Default value
    double y = 0.0;    // Default value
};
```

Now creating `Point p;` automatically sets both coordinates to `0.0`.

❖ 6.1.4. Creating and Using `struct`

Once a `struct` type is defined, you can create variables of that type:

```
Student s1;              // Create a variable of type Student
s1.name = "Alex";        // Assign value to the 'name' member
s1.age = 19;             // Assign value to the 'age' member
s1.gpa = 3.8;            // Assign value to the 'gpa' member
```

There is another way to initialize a `struct` when creating it:

```
Student s2 = {"Jordan", 20, 3.5};   // Initialize all members in order
```

❖ 6.1.5. Accessing Members

To access the data inside a `struct`, you use special operators:

- Dot operator (`.`) - used when you have a normal struct variable.
- Arrow operator (`->`) - used when you have a pointer to a struct.

Using a normal `struct` variable:

```
std::cout << "Year: " << today.year << '\n';   // Dot operator with struct variable
```

Using a pointer to a `struct`:

```
Date* p = &today;                              // p points to the struct 'today'
std::cout << "Month: " << p->month << '\n';     // Arrow operator with pointer
```

The arrow operator (`->`) is just a convenient way of saying: "Follow the pointer to the struct, then access this member."

❖ 6.1.6. Using `typedef` for simpler names

`typedef` is a keyword that lets you give a struct type a shorter, cleaner name:

```
// Define a struct with a typedef to give it the name 'Student'
typedef struct {
    char name[40];   // Array of characters to store the student's name
    int age;         // Integer to store the student's age
    float gpa;       // Floating-point number to store the student's GPA
} Student;

// Create a Student variable and initialize its members
Student s1 = {"Mika", 19, 3.8};   // name="Mika", age=19, gpa=3.8
```

In modern C++, `using` is preferred, but `typedef` is still commonly seen

❖ 6.1.7. Arrays of Structures

Structures can be stored naturally in arrays, making it easy to work with lists of records:


```
// Create an array of 3 Student structs, each initialized with name, age, and GPA
Student roster[3] = {
    {"Ana", 20, 3.7},   // roster[0]
    {"Bo", 19, 3.4},    // roster[1]
    {"Chen", 21, 3.9}   // roster[2]
};

// Loop through the array to access and print each student's name and GPA
for (int i = 0; i < 3; i++) {
    std::cout << roster[i].name    // Access the 'name' member of the i-th Student
              << ": "
              << roster[i].gpa     // Access the 'gpa' member
              << '\n';
}
```

❖ 6.1.8. Nested Structures

A `struct` can contain another `struct`—useful for modeling layered or complex data:

```
// Define a struct to represent time with hours, minutes, and seconds
struct Time {
    int hours, minutes, seconds;
};

// Define a struct to represent a video, which includes a title and a duration (Time)
struct Video {
    std::string title;
    Time duration;  // Nested struct
};

// Create a Video object, initializing the title and the duration
Video v = {"Intro to Structs", {0, 7, 45}};

// Access and print the title and the minutes part of the nested duration
std::cout << v.title
          << ", minutes: "
          << v.duration.minutes
          << '\n';
```


❖ 6.1.9. Input and Output with Structures

Read or print each member individually:

```
Student st;                              // Create a Student variable to hold input

std::cout << "Name age gpa: ";           // Prompt the user for the student's info

// Read values into each member of the struct
// std::cin automatically stores the input in the corresponding members
std::cin >> st.name >> st.age >> st.gpa;

std::cout << st.name << " (" << st.age    // Output the stored values
          << ") GPA=" << st.gpa << '\n';
```

❖ 6.1.10. Memory Basics

A struct's size is at least the sum of its members, but compilers may insert padding to align data efficiently. Reordering members (largest to smallest) can sometimes reduce this padding.

```
std::cout << "Size of Student: "
          << sizeof(Student)   // sizeof gives the number of bytes the struct occupies
in memory
          << '\n';
```

❖ 6.1.11. `struct` vs. `class`

In C++, `struct` and `class` are almost identical. The main difference is:

- Members of a `struct` are *public* by default.
- Members of a `class` are *private* by default.

For simple data grouping, `struct` is usually the clearer and simpler choice. You will learn more about `class` in the lesson on Object-Oriented Programming.

Try these examples for yourself with the demo below!

Demo 6.1: StructuredProgramming/Demos/struct.cpp

```cpp
#include <iostream>
#include <string>

int main() {

    // 1) Basic struct definition
    struct Student {           // Define a new struct type named "Student"
        std::string name;      // Member to store the student's name
        int age;               // Member to store age
        double gpa;            // Member to store GPA
    };

    // Creating a struct variable
    Student s1;
    s1.name = "Alex";  // Assign values to members
    s1.age = 19;
    s1.gpa = 3.8;

    std::cout << "Student: " << s1.name
              << ", Age: " << s1.age
              << ", GPA: " << s1.gpa << '\n';

    // Using another way to initialize a struct
    Student s2 = {"Jordan", 20, 3.5};  // List initialization
    std::cout << "Student: " << s2.name
              << ", Age: " << s2.age
              << ", GPA: " << s2.gpa << '\n';

    // 2) Struct with default member values
    struct Point {
        double x = 0.0;  // Default x
        double y = 0.0;  // Default y
    };

    Point p;  // Automatically x=0.0, y=0.0
    std::cout << "Point coordinates: (" << p.x << ", " << p.y << ")\n";

    // 3) Using pointers with structs
    struct Date {
        int day;
        int month;
        int year;
    };

```

Demo 6.1: StructuredProgramming/Demos/struct.cpp

```
45.       Date today = {25, 11, 2025};
46.       Date* ptr = &today;  // Pointer to the struct
47.
48.       std::cout << "Today (dot operator): " << today.day << "/" << today.month <<
              "/" << today.year << '\n';
49.       std::cout << "Today (arrow operator): " << ptr->day << "/" << ptr->month <<
              "/" << ptr->year << '\n';
50.
51.       // 4) typedef for simpler struct names
52.       typedef struct {
53.           char name[40];  // Fixed-size character array for name
54.           int age;
55.           float gpa;
56.       } StudentTypedef;    // 'StudentTypedef' can now be used as a type
57.
58.       StudentTypedef s3 = {"Mika", 19, 3.8};
59.       std::cout << "StudentTypedef: " << s3.name << ", Age: " << s3.age << ", GPA:
              " << s3.gpa << '\n';
60.
61.       // 5) Array of structs
62.       Student roster[3] = {
63.           {"Ana", 20, 3.7},   // roster[0]
64.           {"Bo", 19, 3.4},    // roster[1]
65.           {"Chen", 21, 3.9}   // roster[2]
66.       };
67.
68.       // Loop through the array
69.       for (int i = 0; i < 3; i++) {
70.           std::cout << "Roster[" << i << "]: "
71.                     << roster[i].name << ", GPA: "
72.                     << roster[i].gpa << '\n';
73.       }
74.
75.       // 6) Nested structs
76.       struct Time {
77.           int hours;
78.           int minutes;
79.           int seconds;
80.       };
81.
82.       struct Video {
83.           std::string title;
84.           Time duration;  // Nested struct
85.       };
86.
```

Demo 6.1: StructuredProgramming/Demos/struct.cpp

```
87.       Video v = {"Intro to Structs", {0, 7, 45}};
88.       std::cout << "Video: " << v.title
89.                 << ", Duration: " << v.duration.hours << "h:"
90.                 << v.duration.minutes << "m:"
91.                 << v.duration.seconds << "s\n";
92.
93.       // 7) Input/Output with structs
94.       Student st;
95.       std::cout << "Enter student name, age, gpa: ";
96.       std::cin >> st.name >> st.age >> st.gpa;  // Read values into struct
97.
98.       std::cout << "You entered: " << st.name
99.                 << " (" << st.age << ") GPA=" << st.gpa << '\n';
100.
101.      // 8) Memory basics
102.      std::cout << "Size of Student struct: "
103.                << sizeof(Student) << " bytes\n";
104.
105.      // 9) Struct vs Class
106.      // Note:
107.      // - struct members are public by default
108.      // - class members are private by default
109.      // Structs are often simpler for grouping data without behavior
110.
111.      return 0;
112.  }
```


6.2. typedef and enum

In C++, *user-defined types* are types that you create yourself to give meaning to data in your program. Two simple but powerful tools for this are `typedef` and `enum`. They make your code easier to read, maintain, and understand.

❖ 6.2.1. `typedef` - Giving Types a New Name

`typedef` lets you create an alias (a new name) for an existing type. It doesn't make a new type; it simply provides a shorter or more meaningful name.

❖ 6.2.2. Why use `typedef`?

- Make long or complex type names shorter.
- Improve code readability.
- Simplify working with structs, pointers, arrays, or templates.

Basic syntax:

```
typedef existing_type NewName; // Create a new name (alias) for an existing type
```


Here is an example of assigning an alias for `unsigned`:

```
typedef unsigned int uint; // 'uint' is now an alias for 'unsigned int'

int main() {
    uint age = 25;          // Declare an unsigned integer using the new alias
    std::cout << "Age: " << age << '\n';  // Print the value
    return 0;
}
```

You can use `typedef` with a `struct`:

```
// Define a struct and give it a shorter name using typedef
typedef struct {
    std::string name;   // Member to store student's name
    int age;            // Member to store student's age
    float gpa;          // Member to store student's GPA
} Student;              // 'Student' can now be used as a type

int main() {
    Student s1 = {"Alex", 20, 3.8};  // Create and initialize a Student
    std::cout << s1.name << ", "      // Access and print the 'name' member
              << s1.age << ", GPA: "  // Access and print the 'age' member
              << s1.gpa << '\n';      // Access and print the 'gpa' member
    return 0;
}
```

In modern C++, you can also use `using` as a more readable alternative:

```
using Student2 = struct { std::string name; int age; float gpa; };
```

You can also give the `struct` and the `typedef` the same name:


```
typedef struct Student {
    int id;             // Student ID
    float gpa;          // Student GPA
} Student;              // Typedef name matches struct tag

Student s;          // Use the typedef name
struct Student t;   // Use the struct tag (same underlying type)
```

❖ 6.2.3. `enum`: Named Integer Constants

An `enum` (short for *enumeration*) lets you define a set of named integer constants. It makes code easier to read and reduces the need to hard code numbers in your code.

❖ 6.2.4. Why use `enum`?

- Give meaningful names to numbers.
- Avoid hardcoding numbers in your code.
- Make code easier to read, maintain, and debug.

- Group related constants together logically.

Basic syntax:

```
enum Color {
    RED,      // 0 by default
    GREEN,    // 1
    BLUE      // 2
};
```

A common application is to use an `enum` to enumerate day names:

```
enum Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };

int main() {
    Day today = Wednesday;                  // Assign a named constant
    std::cout << "Today is day number: "
              << today << '\n';             // Prints 2 (0-based index)
    return 0;
}
```


By default, the first enumerator is `0`, the next is `1`, and so on.

You can also set **specific values** for the constants:

```
enum Color { Red = 1, Green = 5, Blue = 10 }; // Assign specific integer values

int main() {
    Color favorite = Green;                  // Assign the 'Green' value
    std::cout << "Favorite color code: "
              << favorite << '\n';          // Prints 5
    return 0;
}
```

Modern C++ permits you to define an `enum` **class**:

```
enum class ModernColor {RED, GREEN, BLUE};
int main() {
   ModernColor mc = ModernColor::BLUE; // enumeration is Scoped to class name (ModernCol ↵↵
or)
   std::cout << "Modern enumeration for BLUE: " << static_cast<int>(mc) << '\n'; // Cast
 explicitly to int
```

An `enum` class offers the following advantages over using the basic `enum`:

- The `enum` values are strongly typed. The default type is`int`.
- **Type safety** is provided (explicit casts are required).
- **Scoped names** for enumerations avoid potential conflicts with other variables.

❖ 6.2.5. Combining `typedef` and `enum`

You can combine `typedef` and `enum` to create a reusable named type:

```
// Define an enum with typedef to create a new type 'Priority'
typedef enum {
    Low,        // 0
    Medium,     // 1
    High        // 2
} Priority;

int main() {
    Priority task = Medium;                  // Assign an enum constant
    std::cout << "Task priority: "
              << task << '\n';              // Prints 1
    return 0;
}
```


This creates a type `Priority` that can be used like any other type.

❖ 6.2.6. Why `typedef` and `enum` Are Useful Together

- `typedef` makes structs, enums, or other types easier to write and read.
- `enum` gives meaningful names to numbers, improving code clarity.
- Together, they help you write clean, readable, and maintainable code.

❖ 6.2.7. Naming and Style Tips

- Use clear, descriptive type names: Day, Status, TrafficLight, Point.
- Use uppercase for `enum` values: RED, ERROR, MON.
- Consider short prefixes for enums shared across modules to avoid name clashes: e.g., TL_RED.

- Keep `enum` lists cohesive: all values should belong to the same conceptual group.

Demo 6.2: StructuredProgramming/Demos/typedefandenum.cpp

```
1.  #include <iostream>   // Required for std::cout
2.  #include <string>     // Required for std::string
3.
4.  // typedef Examples
5.
6.  // Example 1: Simple type alias
7.  typedef unsigned int uint;  // 'uint' is now an alias for 'unsigned int'
8.
9.  // Example 2: Using typedef with a struct
10. typedef struct {
11.     std::string name;   // Member to store student's name
12.     int age;            // Member to store student's age
13.     float gpa;          // Member to store student's GPA
14. } Student;              // 'Student' can now be used as a type
15.
16. // Example 3: Naming struct and typedef the same
17. typedef struct StudentTag {
18.     int id;             // Student ID
19.     float gpa;          // Student GPA
20. } StudentAlias;         // Typedef name matches struct tag
21.
22. // Modern C++ alternative using 'using'
23. using Student2 = struct {
24.     std::string name;
25.     int age;
26.     float gpa;
27. };
28.
29. // enum Examples
30.
31. // Example 1: Basic enum
32. enum Color { RED, GREEN, BLUE }; // RED=0, GREEN=1, BLUE=2
33.
34. // Example 2: Days of the week
35. enum Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
36.
37. // Example 3: Enumerators with custom values
38. enum CustomColor { Red = 1, GreenCustom = 5, BlueCustom = 10 };
39.
40. // Example 4: Combining typedef and enum
41. typedef enum {
42.     Low,        // 0
43.     Medium,     // 1
44.     High        // 2
```

Demo 6.2: StructuredProgramming/Demos/typedefandenum.cpp

```
45.  } Priority;
46.
47.  // Example 5: Modern C++ enum class
48.  enum class ModernColor {RED, GREEN, BLUE};
49.
50.  int main() {
51.      // Typedef Demo
52.      uint age = 25;  // Using typedef alias for unsigned int
53.      std::cout << "Age: " << age << '\n';
54.
55.      Student s1 = {"Alex", 20, 3.8}; // Create and initialize a Student
56.      std::cout << s1.name << ", "
57.                << s1.age << ", GPA: "
58.                << s1.gpa << '\n';
59.
60.      StudentAlias s2;   // Using struct tag and typedef
61.      s2.id = 101;
62.      s2.gpa = 3.5;
63.      std::cout << "Student ID: " << s2.id << ", GPA: " << s2.gpa << '\n';
64.
65.      Student2 s3 = {"Jordan", 22, 3.9}; // Using modern 'using' typedef
66.      std::cout << s3.name << ", "
67.                << s3.age << ", GPA: "
68.                << s3.gpa << '\n';
69.
70.      // Enum Demo
71.      Color c = BLUE; // Assign a basic enum
72.      std::cout << "Color BLUE has value: " << c << '\n';
73.
74.      Day today = Wednesday; // Assign enum for day
75.      std::cout << "Today is day number: " << today << '\n'; // Prints 2
76.
77.      CustomColor favColor = GreenCustom; // Custom enum value
78.      std::cout << "Favorite color code: " << favColor << '\n'; // Prints 5
79.
80.      Priority task = Medium; // Typedef and enum
81.      std::cout << "Task priority: " << task << '\n'; // Prints 1
82.
83.      ModernColor mc = ModernColor::BLUE; // enumeration is Scoped to class name
           (ModernColor)
84.     std::cout << "Modern enumeration for BLUE: " << static_cast<int>(mc) << '\n';
          // Cast explicitly to int
85.
86.      return 0;
```

Demo 6.2: StructuredProgramming/Demos/typedefandenum.cpp

```
87.  }
```


6.3. Passing Structs to Functions

Now you already know what a `struct` is and how to create and use it from the previous reading. This reading focuses on passing structs to functions in C++, so you can efficiently read, modify, or compute with structured data.

❖ 6.3.1. Ways to Pass a `struct` to Functions

In C++, there are 4 ways to pass a `struct` to a function:

1. **Pass by value** – the function receives a copy of the `struct`. Changes inside the function do not affect the original.
2. **Pass by reference** – the function receives the original `struct`, so modifications affect the original.
3. **Pass by const reference** – the function can read the original `struct` but cannot modify it. Efficient and safe.
4. **Pass by pointer** – the function receives the address of the `struct`. Use `->` to access members.

These methods let you control efficiency and safety depending on whether the function needs to modify the data.

Pass by Value

Passing a `struct` by value means the function receives a copy. Changes inside the function do not affect the original `struct`.

```
// Define a simple Student struct
struct Student {
    std::string name; // Student's name
    int age;          // Student's age
    float gpa;        // Student's GPA
};

// Function receives a copy of the struct
void printStudent(Student s) {
    // Access members using dot operator
    std::cout << "Name: " << s.name
              << ", Age: " << s.age
              << ", GPA: " << s.gpa << '\n';
}

int main() {
    Student s1 = {"Alex", 20, 3.8}; // Create and initialize Student
    printStudent(s1);               // Pass by value
    // Original s1 remains unchanged
    return 0;
}
```


Note:

- `void printStudent(Student s)` tells the compiler that the function expects a **copy** of the struct.
- The function **cannot modify** the original struct.
- The drawback of this technique is that copying large structs may adversely impact performance.

Pass by Reference

Passing by reference allows the function to work with the original `struct`, so changes to the `struct` affect the original data.

```
// Define Student struct
struct Student {
    std::string name;
    int age;
    float gpa;
};

// Function receives a reference to the original struct
void celebrateBirthday(Student& s) {
    s.age += 1; // Original struct updated
    std::cout << "Happy Birthday " << s.name
              << "! You are now " << s.age << " years old.\n";
}

int main() {
    Student s1 = {"Alex", 20, 3.8};
    celebrateBirthday(s1);           // Pass by reference
    std::cout << s1.name << "'s age now: " << s1.age << '\n'; // Original struct changed

    return 0;
}
```


Note:

- `void celebrateBirthday(Student& s)` tells the compiler that the function expects a **reference** to the struct.
- The function **can modify** the struct.

Pass by const Reference

If you want efficiency without allowing modification, use a `const` reference.

```
// Define Student struct
struct Student {
    std::string name;
    int age;
    float gpa;
};

// Function receives a const reference to prevent modification
void showStudent(const Student& s) {
    // Safe read-only access to struct members
    std::cout << s.name << ", " << s.age << ", GPA: " << s.gpa << '\n';
}

int main() {
    Student s1 = {"Alex", 20, 3.8};
    showStudent(s1);  // Pass by const reference
    // s1 is unchanged
    return 0;
}
```

Note the following:

`void showStudent(const Student& s)` informs the compiler that the function expects a **read-only reference** to the struct.

The function receives the **address** of the struct but **cannot modify** the struct due to the `const` keyword.

Pass Using Pointers

Passing a pointer allows the function to modify the original `struct`. Use `->` to access members.

```
// Define Student struct
struct Student {
    std::string name;
    int age;
    float gpa;
};

// Function receives a pointer to the struct
void increaseGPA(Student* s) {
    s->gpa += 0.1f;  // Arrow operator to access member via pointer
    std::cout << s->name << "'s new GPA: " << s->gpa << '\n';
}

int main() {
    Student s1 = {"Alex", 20, 3.8};
    increaseGPA(&s1);  // Pass address of struct
    // s1's GPA is modified
    return 0;
}
```


Items of interest:

- `void increaseGPA(Student* s)` tells the compiler that the function expects a pointer to the struct.
- The function accesses the struct variables using arrow notation (`->`). For example, `s->gpa`.
- The function **can modify** the struct.

❖ 6.3.2. Returning Structs from Functions

Functions can return structs as computed results.

```
// Define Point struct
struct Point {
    int x; // x-coordinate
    int y; // y-coordinate
};

// Function returns a new Point struct
Point addPoints(Point a, Point b) {
    Point sum;
    sum.x = a.x + b.x; // Sum x values
    sum.y = a.y + b.y; // Sum y values
    return sum;        // Return a new struct
}

int main() {
    Point p1 = {2, 3};
    Point p2 = {4, 5};
    Point result = addPoints(p1, p2);  // Receive returned struct
    std::cout << "Sum: (" << result.x << ", " << result.y << ")\n";
    return 0;
}
```


Note: Returning by value copies the `struct`. For very large structs, returning by reference or pointer can improve efficiency.

❖ 6.3.3. Summary Table

Method	Syntax	Effect on Original `struct`	Use Case
Pass by value	`func(Student s)`	No change	Small structs, safe options.
Pass by reference	`func(Student& s)`	Changes original	Modify `struct` efficiently.
Pass by const reference	`func(const Student& s)`	No change	Read-only, efficient.
Pass by pointer	`func(Student* s)`	Changes original	Dynamic structs or arrays

Demo 6.3: StructuredProgramming/Demos/PassStructs.cpp

```cpp
#include <iostream>
#include <string>

// Define a simple Student struct
struct Student {
    std::string name; // Student's name
    int age;          // Student's age
    float gpa;        // Student's GPA
};

// 1) Pass by Value
// Function receives a copy of the struct
void printStudent(Student s) {
    std::cout << "[Pass by value] "
              << s.name << ", " << s.age << ", GPA: " << s.gpa << '\n';
    s.age += 1; // This change affects only the copy, not the original
}

// 2) Pass by Reference
// Function receives a reference to the original struct
void celebrateBirthday(Student& s) {
    s.age += 1; // Original struct is updated
    std::cout << "[Pass by reference] Happy Birthday "
              << s.name << "! Age now: " << s.age << '\n';
}

// 3) Pass by Const Reference
// Function reads struct without modifying it
void showStudent(const Student& s) {
    std::cout << "[Pass by const reference] "
              << s.name << ", " << s.age << ", GPA: " << s.gpa << '\n';
}

// 4) Pass Using Pointers
// Function receives a pointer to the struct
void increaseGPA(Student* s) {
    s->gpa += 0.1f; // Arrow operator to access members
    std::cout << "[Pass by pointer] " << s->name
              << "'s new GPA: " << s->gpa << '\n';
}

int main() {
    // Create and initialize Student
    Student s1 = {"Alex", 20, 3.8};
```

Demo 6.3: StructuredProgramming/Demos/PassStructs.cpp

```
45.       std::cout << "Age:" << s1.age << '\n';
46.       // --- Pass by value ---
47.       printStudent(s1);
48.       std::cout << "Age after pass by value: " << s1.age << '\n'; // Age unchanged
49.
50.       // --- Pass by reference ---
51.       celebrateBirthday(s1);
52.       std::cout << "Age after pass by reference: " << s1.age << '\n'; // Age changed
53.
54.       // --- Pass by const reference ---
55.       showStudent(s1); // Cannot modify inside function
56.
57.       // --- Pass by pointer ---
58.       increaseGPA(&s1); // Pass address of struct
59.       std::cout << "Original GPA after pointer function: " << s1.gpa << '\n';
60.   }
```

Exercise 6: Practice with User-Defined Types and Enums

15 to 30 minutes

In this exercise, you will continue working on the Banking Menu program.

Open `StructuredProgramming\Demos\BankingMenu.cpp`. This file contains your starter code.

❖ E6.1. Requirements

1. Apply structured programming to the Banking Menu app.
2. Create a `typedef`.
3. Create a `struct` to store account data.
4. Define an `enum` within the `struct`.

❖ E6.2. Step-by-step Instructions

1. After the `#include` preprocessor statements, create a `typedef` that gives `std::string` a new data type name of `str`. Use `str` anywhere in your code in place of `std::string`.
2. Create a struct that contains:
   - `int acctNbr`
   - `str holder`
   - `double balance`
3. Within the struct, create an `enum` class named `AcctType` that contains two items, `Checking` and `Savings`.
4. Prompt the user for the account holder name and store the name in the `holder` variable of the struct.
5. Prompt the user for the account type and store the account type in the `enum`.
6. Hard code an account number (e.g., 555666777) in the `acctNbr` variable in the struct.
7. Start with a balance of $0.0.

8. After the user has entered the deposits and withdrawals and has exited the while loop, store the balance in the `struct`.
9. Call a function to print the variables in the `struct`.

Here is sample output:

```
Please enter account holder name:
Stephen
Enter account type (C for Checking or S for Savings):
C
Enter 'w', 'd', 'i', or 'q'
d
Enter amount to deposit:
100
Enter 'w', 'd', 'i', or 'q'
w
Enter amount to withdraw:
75
Enter 'w', 'd', 'i', or 'q'
q
 Log of Withdrawals
                  75.00
  Log of Deposits
                 100.00
                          Bank Account Report
Account Number  555666777
Account Holder  Stephen
Account Type    Checking
Balance         $275.00
```

Solution: StructuredProgramming/Solutions/BankingMenu.cpp

```
1.   #include <array>
2.   #include <iostream>
3.   #include <format>
4.   #include <stdexcept>
5.   #include <string>
6.
7.   typedef std::string str;
8.
9.   const double START_BALANCE = 0.0; // Initial account balance
10.  double balance = START_BALANCE;   // User's account balance
11.  str userInput;
12.
13.  struct Account {
14.      int acctNbr;
15.      str holder;
16.      double balance;
17.
18.      enum class AcctType {
19.          Checking,
20.          Savings
21.      };
22.
23.      AcctType acctType;
24.  };
25.
26.  str displayAcctType(Account::AcctType acctType) {
27.      switch (acctType) {
28.      case Account::AcctType::Checking:
29.          return "Checking";
30.      case Account::AcctType::Savings:
31.          return "Savings";
32.      default:
33.          return "Invalid Account Type";
34.      }
35.  }
36.
37.  void printAccountStruct(Account& account) {
38.      std::cout << std::format("{:^70}\n", "Bank Account Report");
39.      std::cout << std::format("{:<15} {:<20}\n", "Account Number", account.ac ↵↵
             ctNbr);
40.      std::cout << std::format("{:<15} {:<20}\n", "Account Holder", account.holder);
41.      std::cout << std::format("{:<15} {:<20}\n", "Account Type", displayAcctType(ac ↵↵
             count.acctType));
42.      std::cout << std::format("{:<15} ${:<20.2f}\n", "Balance", account.balance);
```

Solution: StructuredProgramming/Solutions/BankingMenu.cpp

```
43.  }
44.
45.  double withdraw(double amount) {
46.      static int nbrOfWithdrawals = 1;
47.      if (amount > balance) {
48.         std::string error_msg = std::format("Withdrawal rejected: amount ${:.2f}
               exceeds balance.\n", amount);
49.          throw std::runtime_error(error_msg);
50.      }
51.      else if (amount <= 0) {
52.         std::string error_msg = std::format("Withdrawal rejected: amount ${:.2f}
               is less than or equal to zero.\n", amount);
53.          throw std::runtime_error(error_msg);
54.      }
55.      else {
56.          std::cout << std::format("Number of withdrawals is now {}\n",
              nbrOfWithdrawals);
57.          if (nbrOfWithdrawals > 3) {
58.              amount += 3;
59.          }
60.          nbrOfWithdrawals++;
61.          return balance -= amount;
62.      }
63.  }
64.
65.  double deposit(double amount) {
66.      if (amount <= 0) {
67.          std::string error_msg = std::format("Deposit rejected: amount ${:.2f}
              is less than or equal to zero.\n", amount);
68.          throw std::runtime_error(error_msg);
69.      }
70.      else {
71.          return balance += amount;
72.      }
73.  }
74.
75.  int main() {
76.      //Allocate memory for the struct
77.      Account account;
78.
79.      std::array<double, 100> depositLog{};
80.      int depositIndex = 0;
81.      std::array<double, 100> withdrawalLog{};
82.      int withdrawalIndex = 0;
83.
```

Solution: StructuredProgramming/Solutions/BankingMenu.cpp

```
84.       account.acctNbr = 555666777;
85.       // prompt user for account holder name
86.       std::cout << "Please enter account holder name: ";
87.       getline(std::cin, userInput);
88.       account.holder = userInput;
89.
90.       bool correct = false;
91.       // prompt user for the account type
92.       while (!correct && userInput[0] != 'Q') {
93.           std::cout << "Enter \'C\' for Checking or \'S\' S for Savings (Q to
              quit): ";
94.           getline(std::cin, userInput);
95.           switch (userInput[0]) {
96.           case 'C':
97.           case 'c':
98.               account.acctType = Account::AcctType::Checking;
99.               correct = true;
100.              break;
101.          case 'S':
102.          case 's':
103.              account.acctType = Account::AcctType::Savings;
104.              correct = true;
105.              break;
106.          case 'Q':
107.          case 'q':
108.              std::cout << "Banking menu will end...\n";
109.              correct = true;
110.              break;
111.          default:
112.             std::cout << "Unknown accout type...please re-enter (Q to quit).\n";
113.          }
114.          if (userInput[0] == 'q' || userInput[0] == 'Q') {
115.              return 0;
116.          }
117.      }
118.
119.      do {
120.          // Display menu options
121.          std::cout << "\n Deposit (d)\n";
122.          std::cout << " Withdraw (w)\n";
123.          std::cout << " Balance (b)\n";
124.          std::cout << " Quit (q)\n\n";
125.          std::cout << "Enter choice: ";
126.          std::getline(std::cin, userInput);
```

Solution: StructuredProgramming/Solutions/BankingMenu.cpp

```
127.          switch (userInput[0]) {
128.              // Handle deposit
129.              case 'd':
130.              case 'D': {
131.                  double amount;
132.                  std::cout << "Enter amount to deposit: ";
133.                  std::getline(std::cin, userInput);
134.                  try {
135.                      amount = std::stod(userInput);
136.                      balance = deposit(amount);
137.                      depositLog[depositIndex++] = amount;
138.                      std::cout << std::format("Deposited ${:.2f}\n", amount);
139.                  }
140.                  catch (std::invalid_argument& ie) {
141.                   std::cout <<  "Deposit amount must be numeric, amount reject  ↵↵
              ed.\n";
142.                  }
143.                  catch (std::runtime_error& re) {
144.                     std::cout << re.what();
145.                  }
146.                  break;
147.              }
148.              // Handle withdrawal
149.              case 'w':
150.              case 'W': {
151.                  double amount;
152.                  std::cout << "Enter withdrawal amount: ";
153.                  std::getline(std::cin, userInput);
154.                  try {
155.                      amount = std::stod(userInput);
156.                      balance = withdraw(amount);
157.                      withdrawalLog[withdrawalIndex++] = amount;
158.                      std::cout << std::format("Withdrew ${:.2f}\n", amount);
159.                  }
160.                  catch (std::invalid_argument& ie) {
161.                   std::cout <<  "Withdrawal amount must be numeric, amount re  ↵↵
              jected.\n";
162.                  }
163.                  catch (std::runtime_error rte) {
164.                      std::cout << rte.what();
165.                  }
166.                  break;
167.              }
168.              // Display balance
```

Solution: StructuredProgramming/Solutions/BankingMenu.cpp

```
            case 'b':
            case 'B':
               std::cout << std::format("Current balance: ${:.2f}\n", balance);
                break;
            // Handle quit
            case 'q':
            case 'Q':
                std::cout << std::format("Final balance: ${:.2f}\n", balance);
                break;
            // Handle invalid input
            default:
              std::cout << "Invalid selection. Please choose a valid option.\n";
        }

    } while (userInput[0] != 'q' && userInput[0] != 'Q');
    // print withdrawal log
    std::cout << std::format("{:^20}\n", "Log of Withdrawals");
    for (auto& withdrawal : withdrawalLog) {
        if (withdrawal == 0) {
            break;
        }
       std::cout << std::format("{:>20}\n", std::format("${:.2f}",withdrawal));
    }
    // print deposit log
    std::cout << std::format("{:^20}\n", "Log of Deposits");
    for (auto& deposit : depositLog) {
        if (deposit == 0) {
            break;
        }
        std::cout << std::format("{:>20}\n", std::format("${:.2f}",deposit));
    }
    // set the balance in the account struct
    account.balance = balance;
    // print account struct
    printAccountStruct(account);
}
```


Conclusion

In this lesson, you learned how to model data with structs, initialize and access members (dot and arrow), create arrays and nested structures, and understand memory basics and the struct vs. class default visibility. You also created clear type aliases with typedef/using and expressed related constants with

enum and enum class. We explained how to pass and return structs effectively—by value, reference, const reference, and pointer—so you can balance safety, clarity, and performance.

You then applied these ideas to the Banking Menu program, where you defined a typedef (str), built an Account struct with an AcctType enum class, handled input/output, tracked deposits and withdrawals, updated state, and printed a formatted report through a function. We guided you to structure your code cleanly, choose appropriate parameter-passing strategies, and use user-defined types to make your program easier to read, maintain, and extend.

LESSON 7
Standard Data Structures

Topics Covered

☑ Sequence containers: vectors and lists and when to use each.

☑ Stacks, queues, and deques with core operations.

☑ Maps and sets for key-based access.

☑ Hash-based maps and sets and their trade-offs.

☑ Using common algorithms like sort and find, with ranges-based usage.

Introduction

In this lesson, you will explore the core C++ Standard Library data structures and algorithms that help you store, organize, and process data efficiently. You'll learn when and how to use dynamic sequence containers like `std::vector` and `std::list`, container adaptors such as `std::stack` and `std::queue`, the flexible `std::deque`, associative containers like `std::map` and `std::set`, and their hash-based versions (`std::unordered_map` and `std::unordered_set`). You'll also practice essential algorithms such as `std::sort` and `std::find`, and see how C++20 Ranges make common operations simpler and clearer.

Throughout the lesson, you will apply these tools in practical mini-projects, for example, calculating totals with a map, and ordering results with sorting. By the end, you will know how to choose the right container for a task, iterate safely, avoid common mistakes, and write clean, efficient code using the Standard Library.

7.1. Vectors and Lists

The C++ Standard Library provides powerful data structures like `std::vector` and `std::list` as modern, flexible alternatives to built-in C-style arrays. Both are known as **containers** because they store a collection of elements.

These containers can grow or shrink as needed, and include helpful member functions for inserting, removing, and accessing elements.

❖ 7.1.1. Using `std::vector`

The `std::vector` is the most commonly used container in C++. It works like a **dynamic-size** array. Unlike fixed C++ arrays, a **vector** can automatically grow and shrink as you add or remove elements, making it safe and efficient for general use.

Key Features of `std::vector`

- Manages memory dynamically, allowing its size to change at runtime.
- Elements are stored **next to each other** in memory, allowing for fast, random access using an index (e.g., `fruits[2]`).
- Adding or removing an element in the middle requires shifting all subsequent elements.

Let's take a look at an example:

```
#include <vector>
#include <iostream>
#include <string>

int main() {
    // Create a vector
    std::vector<std::string> fruits = { "Apples", "Pears", "Bananas", "Oranges" };
    std::cout << "Vector size is: " << fruits.size() << "\n";

    // Iterate through the vector
    for (auto& fruit : fruits) {
        std::cout << fruit << "\n";
    }

    fruits.push_back("Lemons");                 // add item at end
    fruits.insert(fruits.begin(), "Limes");     // add item at beginning

    // Remove the 2nd item (which is now "Apples")
    fruits.erase(fruits.begin() + 1);

    std::cout << "\nVector contents in reverse:\n";
    for (auto iter = fruits.rbegin(); iter != fruits.rend(); ++iter) {
        std::cout << *iter << "\n";
    }
}
```


Note the following:

- `std::vector<std::string>` is a template declaring that this vector stores strings.
- The vector is initialized with 4 fruits.
- `auto& fruit` deduces the element type and avoids unnecessary copying.
- The `size()` function returns the number of elements in the vector.
- Use `push_back()` function to add an item at the end of a vector.
- Use `insert()` function to insert anywhere in the vector.
- Use `erase()` function to remove an element at a specific position.
- Use `fruits.clear()` if you want to remove all items from the vector.
- `rbegin()` and `rend()` allow reverse iteration
    - `rbegin()` returns last element.

- `rend()` returns one before the first element.

Output

```
Vector size is: 4
Apples
Pears
Bananas
Oranges

Vector contents in reverse:
Lemons
Oranges
Bananas
Pears
Limes
```

❖ 7.1.2. Using `std::list`

The `std::list` container implements a doubly linked list. Instead of storing elements contiguously, each element (called a **node**) stores the actual data plus a pointer to the next node and a pointer to the previous node.

Key Features of `std::list`

- **Non-Contiguous Memory:** Elements are scattered across the memory, connected by pointers.
- **Fast Insertion/Deletion:** Adding or removing elements is extremely fast (constant time, O(1)) because you only need to update a few pointers, not shift the entire list.
- **Slow Random Access:** You cannot jump directly to the 5th element; you must start at the beginning and traverse the pointers one by one. Accessing the 5th element is much slower than in a vector.
- **Dedicated Front/Back Functions:** Has convenient functions for manipulation at the ends.

Here is an example of storing the fruits in a list:

```
#include <list>
#include <iostream>
#include <string>

int main() {
    std::list<std::string> fruitsList = { "Apples", "Pears", "Bananas", "Oranges" };
    std::cout << "List size is: " << fruitsList.size() << "\n";

    std::cout << "List contents at start:\n";
    for (auto& fruit : fruitsList) {
        std::cout << fruit << "\n";
    }

    fruitsList.push_back("Lemons");  // add at end
    fruitsList.push_front("Limes");  // add at beginning

    std::cout << "\nList contents in reverse with fruits added:\n";
    for (auto iter = fruitsList.rbegin(); iter != fruitsList.rend(); ++iter) {
        std::cout << *iter << "\n";
    }
}
```


Note the following:

- Iterating through a list uses the same syntax as iterating through a vector.
- Use the `push_back()` function to add an element at the end of the list.
- Use the `push_front()` function to add an element to the beginning of the list.
- Like vectors, lists support `rbegin()` and `rend()` for reverse iteration.

Output

```
List size is: 4
List contents at start:
Apples
Pears
Bananas
Oranges

List contents in reverse with fruits added:
Lemons
Oranges
Bananas
Pears
Apples
Limes
```

❖ 7.1.3. Key Differences

The choice between `std::vector` and `std::list` depends entirely on how you plan to access and modify the data.

1. Underlying Structure

- `std::vector`: Implements a dynamic array. Its elements are stored in contiguous memory (right next to each other), which is fast for moving through data sequentially.
- `std::list`: Implements a doubly linked list. Its elements are scattered across memory, with each element (node) containing **pointers** to the next and previous elements.

2. Accessing Elements (Speed)

- `std::vector`: Accessing an element at any position using an index (e.g., `myVector[i]`) is extremely fast due to the contiguous memory layout.
- `std::list`: Accessing an element by position is slow. You must start at the beginning and traverse the internal pointers one by one until you reach the desired element.

3. Insertion and Deletion (Speed)

- `std::vector`: Inserting or deleting elements in the middle is Slow. The vector must shift all subsequent elements in memory to maintain contiguity.

- `std::list`: Inserting or deleting elements anywhere (even the middle) is fast. Only the pointers of the neighboring elements need to be updated, without moving any data.

4. When to Use

- Use `std::vector` when you need frequent **fast random access** and your container size changes infrequently or only at the end. It's the standard, general-purpose choice.
- Use `std::list` when you need frequent **insertions or deletions** in the middle of the container, and you rarely need to jump directly to a specific element by index.

❖ 7.1.4. Demo: Using `std::vector` and `std::list`

This program demonstrates the core functionality of `std::vector` and `std::list`.

Demo 7.1: StandardDataStructures/Demos/VectorsAndListDemo.cpp

```
1.   #include <vector>
2.   #include <list>
3.   #include <string>
4.   #include <iostream>
5.
6.   int main() {
7.       // Vector Demo
8.       std::vector<std::string> fruits = { "Apples", "Pears", "Bananas", "Oranges"
             };
9.
10.      std::cout << "Vector size: " << fruits.size() << "\n";
11.      std::cout << "Vector contents:\n";
12.      for (auto& fruit : fruits) {
13.          std::cout << fruit << "\n";
14.      }
15.
16.      fruits.push_back("Lemons");              // add at end
17.      fruits.insert(fruits.begin(), "Limes"); // add at beginning
18.      fruits.erase(fruits.begin() + 1);       // remove item
19.
20.      std::cout << "Vector in reverse:\n";
21.      for (auto iter = fruits.rbegin(); iter != fruits.rend(); ++iter) {
22.          std::cout << *iter << "\n";
23.      }
24.
25.      // List Demo
26.      std::list<std::string> fruitsList = { "Apples", "Pears", "Bananas", "Oranges"
             };
27.
28.      std::cout << "\nList size: " << fruitsList.size() << "\n";
29.      std::cout << "List contents:\n";
30.      for (auto& fruit : fruitsList) {
31.          std::cout << fruit << "\n";
32.      }
33.
34.      fruitsList.push_back("Lemons");  // add at end
35.      fruitsList.push_front("Limes");  // add at beginning
36.
37.      std::cout << "List in reverse:\n";
38.      for (auto iter = fruitsList.rbegin(); iter != fruitsList.rend(); ++iter) {
39.          std::cout << *iter << "\n";
40.      }
41.      std::cout << "\nEnd of Demo!";
42.
```

Demo 7.1: StandardDataStructures/Demos/VectorsAndListDemo.cpp

```
43.      return 0;
44.  }
```


7.2. Stacks, Queues, and Deques

❖ 7.2.1. Container Adaptors: Stacks and Queues

Container adaptors are specialized classes that provide restricted but focused functionality over existing container types. Two essential adaptors in the C++ Standard Library are `std::stack` and `std::queue`. These adaptors typically restrict operations on the `std::deque` (pronounced "deck") container class.

❖ 7.2.2. Stacks (`std::stack`)

A stack follows the **Last In, First Out** (LIFO) principle, meaning the element added most recently is the first one available to be removed.

Think of a stack of plates. You can only add a plate to the top, and you can only take a plate from the top.

Stacks are commonly used for managing function calls, undo operations, expression evaluation, and more. In C++, calls to functions are managed using a stack, ensuring the latest function call finishes before returning to the previous one.

Let's take a look at a simple example:

```
#include <stack>
#include <iostream>

int main() {
    // Create a stack of messages
    std::stack<std::string> messageStack;

    // Push items onto the stack (LIFO behavior)
    messageStack.push("Market product.");
    messageStack.push("Get parts and assemble product.");
    messageStack.push("Design product.");

    std::cout << "Printing messages on the stack:\n";

    // Pop and print until empty
    while (!messageStack.empty()) {
        std::cout << messageStack.top() << '\n';
        messageStack.pop(); // remove the top item
    }
}
```


Note the following:

- The output confirms LIFO as the last message placed on the stack (`"Design product."`) is printed first.
- `#include <stack>` is required to use the stack adaptor.
- `std::stack<std::string>` defines a stack where each element is of type `std::string`.
- The `push()` function **inserts** an item onto the top of the stack.
- The `pop()` function **removes** the top (most recently inserted) element from the stack.
- The `top()` function **returns** a reference to the top element (the one to be removed next) without removing it.
- The `empty()` function returns `true` if the stack contains no items, `false` otherwise.

Here is the output from the program:

Output

```
Printing messages on the stack:
Design product.
Get parts and assemble product.
Market product.
```

❖ 7.2.3. Queues (`std::queue`)

A queue operates on the **First In, First Out** (FIFO) principle, ensuring elements are processed in the exact order they were added.

Think of a line of customers at a cash register lane in a grocery store. The person who arrives first is served first.

Here is a simple example of a queue using the messages from the stack example:

```
#include <queue>
#include <iostream>

int main() {
    std::queue<std::string> messageQueue;

    // Push items onto the queue (FIFO behavior)
    messageQueue.push("Design product.");
    messageQueue.push("Get parts and assemble product.");
    messageQueue.push("Market product.");

    std::cout << "\nPrinting messages on the queue:\n";

    while (!messageQueue.empty()) {
        std::cout << messageQueue.front() << '\n';
        messageQueue.pop(); // remove the front element
    }
}
```


Note the following:

- The output confirms FIFO as the first message pushed on the queue (`"Design product."`) is printed first.
- `#include <queue>` is required to use the queue adaptor.

- `std::queue<std::string>` is a reference to the queue class template. Each item in the queue will be of type `std::string`.
- The `push()` function inserts an item at the back of the queue.
- The `front()` function returns a reference to the element at the front of the queue (the one that has been waiting the longest).
- The `back()` function returns a reference to the element at the back of the queue (the one most recently inserted).
- The `pop()` function removes the element from the front of the queue.

And here is the output:

Output

```
Printing messages on the queue:
Design product.
Get parts and assemble product.
Market product.
```


❖ 7.2.4. Deques (`std::deque`)

The `std::deque` implements a **deque**, or double-ended queue. Deque is a flexible container class that allows efficient insertion and removal of elements at both the front and the back. This makes it suitable as the underlying container for both `std::stack` and `std::queue`.

Therefore, a deque can behave like a stack or a queue.

Once again, let's consider a simple example using the message data:

```
#include <deque>
#include <iostream>

int main() {
    std::deque<std::string> messageDeque;

    // Use deque like a queue
    messageDeque.push_back("Design product.");
    messageDeque.push_back("Get parts and assemble product.");
    messageDeque.push_back("Market product.");

    std::cout << "\nPrinting deque as a queue:\n";
    while (!messageDeque.empty()) {
        std::cout << messageDeque.front() << '\n';
        messageDeque.pop_front(); // remove from front (queue behavior)
    }

    // Use deque like a stack
    messageDeque.push_front("Market product.");
    messageDeque.push_front("Get parts and assemble product.");
    messageDeque.push_front("Design product.");

    std::cout << "\nPrinting deque as a stack:\n";
    for (const auto& msg : messageDeque) {
        std::cout << msg << '\n';
    }
}
```


Note the following:

- `messageDeque` is processed first as a queue, and then as a stack.
- `#include <deque>` is required to use the deque container.
- `std::deque<std::string>` is a reference to the deque class template. Each item in the deque will be of type `std::string`.
- `push_front()` inserts an item at the front of the deque (useful for stack-like behavior).
- `pop_front()` removes the front item in the deque.
- `push_back()` pushes an item at the back, or end, of the queue.
- `pop_back()` can be used to remove the item from the back.
- `clear()` will delete all items from the deque.

Here is the output:

Output

```
Printing deque as a queue:
Design product.
Get parts and assemble product.
Market product.

Printing deque as a stack:
Design product.
Get parts and assemble product.
Market product.
```

❖ 7.2.5. Key Points to Remember

Stack (Last In, First Out) LIFO

- A stack works like a pile of books. The last book placed on top is the first one you take off.
- Only the top item is accessible as you can only look at or remove the item at the top. This makes stacks simple but limited on purpose.
- `push()` adds to the top and `pop()` removes from the top. You always add and remove from one end, which keeps the structure predictable.
- **Useful** when the most recent thing matters the most.
- Examples:
    - Undo/redo in editors
    - Browser "back" button
    - Function call stack
- LIFO makes stacks perfect for **reversing** things because the last item you put in comes out first, stacks naturally reverse order.

Queue (First In, First Out)

- A queue works like people waiting in line. The first person to join the line is the first one served.
- Items come out in the exact order they arrived. No jumping ahead and everything stays fair and predictable.

- `push()` adds to the back and `pop()` removes from the front. You add new items to the **end** and remove from the **front**.
- Used when tasks must happen in order.
- Examples:
    - Print jobs
    - Customer support tickets
    - Tasks waiting to run in the background
- Queues prevent skipping. This ensures no item is processed earlier than it should be.

Deque (Double-Ended Queue)

A deque (pronounced "deck") is like a train with doors at both ends. Passengers can enter or exit from the front or the back.

- You can add or remove from both the front and the back. This makes it more flexible than a stack or queue.
- Can act like either a stack or a queue:
    - Use `push_back`/ `pop_back`, it works like a **stack**
    - Use `push_back` / `pop_front`, it works like a **queue**
- Useful when you need fast access from both sides.
    - Examples:
        - Sliding window algorithms
        - Managing browser history forward/backward
        - Any situation where elements must be handled from both directions

Feel free to run the following demo file for more practice with stacks, queues, and deques.

Demo 7.2: StandardDataStructures/Demos/StackAndQueueDemo.cpp

```cpp
#include <iostream>
#include <stack>
#include <queue>
#include <deque>

int main() {
    // STACK (LIFO)
    std::stack<int> numberStack;

    numberStack.push(10);  // first
    numberStack.push(20);  // second
    numberStack.push(30);  // last pushed, first out

    std::cout << "STACK (LIFO):\n";
    while (!numberStack.empty()) {
        std::cout << numberStack.top() << "\n"; // prints last added item first
        numberStack.pop();
    }

    // QUEUE (FIFO)
    std::queue<int> numberQueue;

    numberQueue.push(100); // first
    numberQueue.push(200); // second
    numberQueue.push(300); // third

    std::cout << "\nQUEUE (FIFO):\n";
    while (!numberQueue.empty()) {
        std::cout << numberQueue.front() << "\n"; // prints oldest added item
         first
        numberQueue.pop();
    }

    // DEQUE (both ends open)
    std::deque<int> numberDeque;

    // Use it like a queue (push back, pop front)
    numberDeque.push_back(1);
    numberDeque.push_back(2);
    numberDeque.push_back(3);

    std::cout << "\nDEQUE as QUEUE:\n";
    while (!numberDeque.empty()) {
        std::cout << numberDeque.front() << "\n";
```

Demo 7.2: StandardDataStructures/Demos/StackAndQueueDemo.cpp

```
44.           numberDeque.pop_front();
45.       }
46.
47.       // Use it like a stack (push front, pop front)
48.       numberDeque.push_front(7);
49.       numberDeque.push_front(8);
50.       numberDeque.push_front(9);
51.
52.       std::cout << "\nDEQUE as STACK:\n";
53.       while (!numberDeque.empty()) {
54.           std::cout << numberDeque.front() << "\n";
55.           numberDeque.pop_front();
56.       }
57.
58.       return 0;
59.   }
```


7.3. Maps and Sets

❖ 7.3.1. Maps and Sets: Associative Containers

Maps and sets are associative containers. **Associative containers** are C++ structures that store elements organized by a **key**. This means they store data in a way that lets you quickly look up a value using a key. They are optimized for fast retrieval, insertion, and removal based on that key. Both containers keep their elements **sorted** internally, and both require **unique keys**.

The two most common are `std::map` and `std::set`.

Key Terms
Associative Container: A container where elements are accessed using a unique identifier (a key), not a numerical index.

Key-Value Pair: The fundamental element of a map, consisting of a unique **Key** used for lookup and a **Value** that holds the associated data.

Binary Search Tree: The common underlying structure for `std::map` and `std::set`, which keeps the data sorted by key and allows for very fast lookups.

❖ 7.3.2. Understanding `std::map`

The `std::map` stores unique **key-value pairs**. Each key is unique, located through binary search, and the map automatically keeps everything sorted by key. Maps are useful for scenarios where data needs to be retrieved quickly using specific identifiers.

It functions like a dictionary or lookup table. Given a key, you can quickly find its associated value. Data in a map is always kept sorted based on the key.

A good real-life example is a fruit stand storing the name of each fruit and its quantity.


```
#include <map>
#include <string>
#include <iostream>

int main() {
    std::map<std::string, int> fruitsMap = {
        {"Apples", 10},
        {"Pears", 2},
        {"Bananas", 3},
        {"Oranges", 9}
    };
    std::cout << "Map with fruit names and quantities:\n";

    for (auto& [fruit, qty] : fruitsMap) {
        std::cout << "Fruit: " << fruit << " , Quantity: " << qty << "\n";
    }
    fruitsMap["Lemons"] = 20;   // Add "Lemons" with quantity=20
    fruitsMap["Apples"] = 8;   // Update "Apples" with quantity=8
    fruitsMap["Bananas"] -= 2;  // Decrease "Bananas" by quantity=2
    fruitsMap.erase("Pears");   // Remove "Pears"
}
```

Note the following

- `std::map<std::string, int>` is a template specifying the **key** is `std::string` and the **value** is `int`.
- The map is initialized with `{Key, Value}` pairs inside the curly braces.
- When iterating through the map using a range-based loop, `auto& [fruit, qtyOnHand]` unpacks the key and value simultaneously.
- You can access or update a value using the key: `fruitsMap["Lemons"] = 20;`.
- You remove an entry by calling the `erase()` member function with the key as the argument.

Output

```
Map with fruit names and quantities:
Fruit: Apples , Quantity: 10
Fruit: Bananas , Quantity: 3
Fruit: Oranges , Quantity: 9
Fruit: Pears , Quantity: 2
```


❖ 7.3.3. Understanding `std::set`

The `std::set` container stores unique elements, similar to a mathematical set. It operates like a **key-only map** container. It only stores the key, and that key must be unique.

Sets are ideal for scenarios where you need to check quickly whether an item exists in a collection. It is useful when you only care about whether something exists, not how many times it appears.

Here is an example to store our fruits in a set:

```
#include <set>
#include <string>
#include <iostream>

int main() {
    std::set<std::string> fruitsSet = {
        "Apples", "Pears", "Bananas", "Oranges"
    };
    std::cout << "Set with fruits:\n";
    for (auto& fruit : fruitsSet) {
        std::cout << fruit << "\n";
    }
    fruitsSet.insert("Lemons");  // Add "Lemons"

 // Try adding "Apples", it's already in the set so the insert will fail
    auto result = fruitsSet.insert("Apples");

    if (result.second) {
        std::cout << "Apples inserted!\n";
    } else {
        std::cout << "Apples are already in the set!\n";
    }
    fruitsSet.erase("Bananas");  // Remove "Bananas"
}
```


Note the following:

- Iteration through a set is identical to iterating through a vector or list using a range-based `for` loop. The output is always **sorted**.
- A set contains only keys with no associated values.
- All elements must be unique.
- Use `insert()` to add an element to the set. It returns a `pair` where the second value is a `bool` showing success or failure.
- You access the second value of the pair (`result.second`), which is a `bool`, to check if the insertion was successful (`true`) or failed (`false`) because the key already existed.
- If the `bool` is `false`, the element already exists.
- `erase()` removes an element if it exists.

Output

```
Set with fruits:
Apples
Bananas
Oranges
Pears
Apples are already in the set!
```

❖ 7.3.4. Key Points to Remember

- ### Maps store key–value pairs

  A map allows you to search using a specific key (like a name or ID) and retrieve an associated value. This makes maps perfect for storing small databases, counts, or labeled data.

- ### Sets store only unique keys

  A set ensures there are no duplicates, making it useful when you only need to track the existence of items.

- ### Both containers automatically sort their elements

  The internal sorting makes lookups fast, but it also means the order will not match the order of insertion.

- ### Accessing map elements may create new keys

  Using `map[key]` inserts the key if it doesn't already exist. We often don't expect this behavior.

- ### Iteration is simple and readable

  - Maps allow unpacking into `[key, value]`.
  - Sets behave like iterating over a sorted list of unique items.

- ### Both containers support `erase()` for removing elements

  Calling `erase(key)` removes the matching element immediately.

❖ 7.3.5. Common Mistakes

- ### Expecting insertion order to be preserved

  Maps and sets always sort their elements, so printing them shows the sorted result, not the insertion order.

- ### Accidentally creating map entries

  Using `map[key]` for checking values can create new entries unexpectedly. If you only want to check whether a key exists, `map.contains(key)` is safer.

- ### Misunderstanding the return value of `set.insert()`

  The returned pair can confuse many people. The important part is the `bool`:

  - `true` means element added
  - `false` means element already existed

- ### Forgetting required headers

  Missing `<map>` or `<set>` leads to compilation errors.

- ### Expecting sets to allow duplicates

  A set will never store the same item twice. We sometimes think a second insert overwrites, but it does nothing.

❖ 7.3.6. Demo: Using `std::map` and `std::set`

This demo file illustrates the core unique behaviors of both `std::map` (Key-Value access) and `std::set` (Key-only access).

Demo 7.3: StandardDataStructures/Demos/MapAndSetDemo.cpp

```cpp
#include <iostream>
#include <map>
#include <set>
#include <string>

int main() {
    // ---Map Demo---
    std::cout << "--- Map Demo ---\n";
    std::map<int, std::string> employeeMap;

    // Insert values (key = ID, value = name)
    employeeMap[101] = "Sarah";
    employeeMap[205] = "Mark";

    // Updating existing key
    employeeMap[101] = "Sarah J.";
    std::cout << "ID 101 Name: " << employeeMap[101] << "\n";


    // ---Set Demo---
    std::cout << "\n--- Set Demo ---\n";
    std::set<double> uniqueNumbers;

    // Insert elements
    uniqueNumbers.insert(3.14);
    uniqueNumbers.insert(1.618);

    // Attempt to insert a duplicate
    auto insertResult = uniqueNumbers.insert(3.14);
    std::cout << "Total unique numbers: " << uniqueNumbers.size() << "\n";

    // Check if duplicate insertion failed
    if (insertResult.second == false) {
        std::cout << "3.14 was not added again (already exists).\n";
    }

    // Check if a value exists
    if (uniqueNumbers.count(1.618)) {
        std::cout << "1.618 is in the set.\n";
    }
    return 0;
}
```

7.4. Unordered Containers

Unordered containers like `std::unordered_map` and `std::unordered_set` use a technique called **hashing** to organize data. Unordered containers give us a faster way to store and find data when order does not matter.

They use **hash tables**, which allow very quick lookups, insertions, and deletions (on average constant time). They are similar to `std::map` and `std::set`, but do not keep elements sorted.

What is Hashing

A process where an input value (the Key) is run through a mathematical function (the **hashfunction**) to produce a fixed-size number (the **hash code**). This code points directly to the element's storage location.

This is why operations are very fast. The container jumps directly to a slot rather than searching through sorted keys.

Real-life example: Think of hashing like placing items in labeled lockers. If you know a locker number, you can directly reach it instead of searching all lockers one by one.

❖ 7.4.1. Understanding `std::unordered_map`

An `unordered_map` stores data in unique **key–value pairs**, similar to a dictionary. It is the fast, unsorted alternative to `std::map`. It does not store keys in order, instead, elements are placed in buckets based on their hash codes.

Use `unordered_map` when:

- you need fast lookups
- you don't care about the order of keys
- keys are unique

Key Features of `std::unordered_map`:

- **Hash-Based Lookup:** Instead of searching through a sorted structure, the container calculates the hash code of the key to jump directly to the data's location.
- **No Order:** The order of elements is unpredictable and will likely change as new elements are added or removed.

Example

```
#include <iostream>
#include <unordered_map>
#include <string>

int main() {
    // Key: fruit name (string), Value: quantity on hand (int)
    std::unordered_map<std::string, int> fruitsMap = {
        { "Apples", 10}, {"Pears", 2}, {"Bananas", 3}
    };

    // Fast lookup using the key
    fruitsMap["Oranges"] = 9;

    // Update value for an existing key
    fruitsMap["Apples"] = 8;
    std::cout << "Quantity of Bananas: " << fruitsMap["Bananas"] << "\n";

    std::cout << "Unordered Map Contents:\n";
    for (auto const& [key, val] : fruitsMap) {
        std::cout << key << ": " << val << "\n";
    }

    return 0;
}
```


Note the following:

- `std::unordered_map<std::string, int>` is the template used, similar to `std::map`.
- Access and modification use the same syntax as `std::map`, using the key within square brackets (e.g., `fruitsMap["Oranges"]`).
- Keys must be unique.
- Internally uses hashing, so operations are fast on average.
- Use it when quick access matters more than sorted order.

Output

```
Quantity of Bananas: 3
Unordered Map Contents:
Bananas: 3
Pears: 2
Oranges: 9
Apples: 8
```

❖ 7.4.2. Understanding `std::unordered_set`

The `std::unordered_set` stores a collection of unique elements (keys only, no values) with no particular order. It is the fast, unsorted alternative to `std::set`.

Use `unordered_set` when you need:

- a list of unique items,
- fast checking if something exists,
- fast insert and delete.

Key Features of `std::unordered_set`:

- **Unique Elements:** Guarantees that every stored element is distinct.
- **Ideal Use:** Perfect for checking membership ("Is this item already in my list?") very quickly.

Example

```
#include <iostream>
#include <unordered_set>
#include <string>

int main() {
    std::unordered_set<std::string> uniqueNames = { "Alice", "Bob", "Charlie" };

    // Fast insertion
    uniqueNames.insert("David");

    // Attempt to insert a duplicate (will fail quickly)
    uniqueNames.insert("Alice");

    std::cout << "--- Search Operation ---\n";
    // Checking for existence is very fast (count returns 1 or 0)
    if (uniqueNames.count("Bob")) {
        std::cout << "Bob is present in the set.\n";
    }

    std::cout << "Total unique names before removal: " << uniqueNames.size() <<"\n";
    // Removal is also very fast
    uniqueNames.erase("Charlie");
    std::cout << "Total unique names after removal : " << uniqueNames.size() << "\n";

    return 0;
}
```


Note the following:

- Sets only require the key type (`std::string` in this case).
- The `count()` function is the fastest way to check if an element exists, returning `1` (if found) or `0` (if not found).
- The `insert()` and `erase()` operations are optimized for speed, leveraging the hash table structure.
- Insertions of duplicates fail silently (the element just does not get added).

Output

```
--- Search Operation ---
Bob is present in the set.
Total unique names before removal: 4
Total unique names after removal : 3
```

❖ 7.4.3. Key Points to Remember

- Unordered containers do not maintain order.
- They use **hash tables**, giving fast average performance.
- `unordered_map` stores **key–value pairs**.
- `unordered_set` stores **unique keys** only.
- Faster than `map` and `set` when you don't need sorted data.

❖ 7.4.4. Demo: Using Unordered Containers

This demo file illustrates the core usage and unordered nature of both hash-based containers.

Note: The order of the city names in the output may vary depending on the compiler's hash function.

Demo 7.4: StandardDataStructures/Demos/UnorderedMapAndSet.cpp

```
1.  #include <iostream>
2.  #include <unordered_map>
3.  #include <unordered_set>
4.  #include <string>
5.
6.  int main() {
7.      // ---Unordered Map Demo---
8.      std::cout << "--- Unordered Map ---\n";
9.
10.     // Key=City (string), Value=Population (int)
11.     std::unordered_map<std::string, int> cityPopulations = {
12.         { "New York", 8000000 },
13.         { "London", 9000000 }
14.     };
15.     cityPopulations["Tokyo"] = 14000000;
16.
17.     // Access is quick, but iteration order is unpredictable
18.     for (auto const& [city, pop] : cityPopulations) {
19.         std::cout << city << ": " << pop << "\n";
20.     }
21.     // ---Unordered Set Demo---
22.     std::cout << "\n--- Unordered Set ---\n";
23.     std::unordered_set<int> uniqueIDs;
24.
25.     uniqueIDs.insert(501);
26.     uniqueIDs.insert(100);
27.     uniqueIDs.insert(501); // Duplicate attempt
28.
29.     std::cout << "Total unique IDs: " << uniqueIDs.size() << "\n"; // Output: 2
30.
31.     // Fast check for membership
32.     if (uniqueIDs.count(100)) {
33.         std::cout << "ID 100 found successfully.\n";
34.     }
35.
36.     return 0;
37. }
```

7.5. Common Algorithms: sort, find, Ranges

C++ provides a wide variety of algorithms that help you process collections of data, like arrays, vectors, lists, maps, and sets. All of these collections are **ranges**. Ranges are basically **pairs of iterators,** one pointing to the start of the container and one pointing just past the end.

Some of the most frequently used algorithms in C++ programming are `sort` and `find`.

❖ 7.5.1. Sorting a Vector

Sorting is one of the most common operations when working with data. C++ makes it easy using the `std::sort` algorithm, defined in the `<algorithm>` header. The `std::sort` algorithm provides a fast and efficient way to rearrange elements within a specified range.

Let's consider an example where we sort a vector in ascending sequence:

Example: Ascending Sort

```
#include <iostream>
#include <vector>
#include <algorithm>  // use for sort
#include <string>

int main() {
    std::vector<std::string> fruitsToSort = { "Pears", "Bananas", "Oranges", "Apples",
"Lemons"};

    // Sorts the entire range in ascending order (A-Z)
    std::sort(fruitsToSort.begin(), fruitsToSort.end());

    std::cout << "Sorted fruits ascending:\n";
    for (const auto& fruit : fruitsToSort) {
        std::cout << fruit << "\n";
    }
    return 0;
}
```

Note the following:

- The required header for `std::sort` is `<algorithm>`.
- `fruitsToSort.begin()` returns an iterator pointing to the first element.
- `fruitsToSort.end()` returns an iterator pointing immediately after the last element.
- The operation is performed in place, meaning the original data structure (`fruitsToSort`) is permanently modified, and the initial ordering is lost.
- By default, `std::sort` arranges elements in **ascending order**.

Here is the output:

```
Sorted fruits ascending:
Apples
Bananas
Lemons
Oranges
Pears
```

To sort the vector in **descending** order, you must provide a **predicate** as the third argument.

Example: Descending Sort

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional> // Required for std::greater

int main() {
    std::vector<std::string> fruitsToSort = { "Pears", "Bananas", "Oranges", "Apples",
"Lemons" };
    // Sorts the range in descending order using the std::greater predicate
    std::sort(fruitsToSort.begin(), fruitsToSort.end(), std::greater<>{});

    std::cout << "Sorted fruits descending:\n";
    for (const auto& fruit : fruitsToSort) {
        std::cout << fruit << "\n";
    }
    return 0;
}
```

Items of interest:

- `std::greater<>{}` is a **predicate** that tells `std::sort` to order elements from largest to smallest.
- The vector is updated in place.

Output

```
Sorted fruits descending:
Pears
Oranges
Lemons
Bananas
Apples
```


❖ 7.5.2. Finding an Element

Sometimes you need to locate one or more occurrences of an element in a container. C++ provides the `std::find` algorithm for this purpose. The `std::find` algorithm is used to locate the first occurrence of a specific value within a range.

To find all occurrences of an element, you must repeatedly call `std::find`, starting the search from the point where the previous match was found.

Example: Find All Occurrences

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <string>

int main() {
    std::vector<std::string> fruits = { "Pears", "Bananas", "Oranges", "Apples", "Lemons",
 "Bananas" };
    auto iter = fruits.begin();
    bool found = false;

    while ((iter = std::find(iter, fruits.end(), "Bananas")) != fruits.end()) {
        std::cout << "Found " << *iter << " at index "
                  << std::distance(fruits.begin(), iter) << "\n";
        found = true;
        ++iter; // move to the next element to continue searching
    }
    if (!found) {
        std::cout << "Bananas not found.\n";
    }
    return 0;
}
```


Note the following:

- `iter` is an iterator pointing to the **current element** being checked. `iter` is initially set to `fruits.begin()`, pointing to the first element.
- `std::find` takes three arguments:
    1. Start iterator (`iter` in this case)
    2. End iterator (`fruits.end()`)
    3. The value to search for (`"Bananas"`)
- `std::distance(fruits.begin(), iter)` computes the location (index) of the found element by counting how many steps the iterator `iter` is away from the starting iterator (`fruits.begin()`).
- Incrementing the iterator (`++iter`) ensures we continue searching past the current found element.

Output

```
Found Bananas at index 1
Found Bananas at index 5
```

❖ 7.5.3. Ranges in C++20

C++20 introduced the **Ranges Library**, which makes working with algorithms simpler and more readable. Ranges allow you to apply algorithms directly to containers without manually specifying iterators.

Features of Range Algorithms

- **Simplified Syntax:** Algorithms like `std::ranges::sort` accept the container directly.
- **Namespace:** Range algorithms are located within the `std::ranges::` namespace.

We can replace the iterator-based sort with the cleaner range-based sort:

Example: Range-Based Sort


```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

int main() {
    std::vector<std::string> fruitsToSort = { "Pears", "Bananas", "Oranges", "Apples",
"Lemons" };

    // Ascending sort (Pass container directly)
    std::ranges::sort(fruitsToSort);
    std::cout << "Ranges Ascending Sort: Done.\n";

    // Descending sort (Pass container and the predicate)
    fruitsToSort = { "Pears", "Bananas", "Oranges", "Apples", "Lemons" }; // Reset for
comparison
    std::ranges::sort(fruitsToSort, std::greater<>{});

    std::cout << "Ranges Descending Sort: Done.\n";
    return 0;
}
```

Items of interest:

- The Ranges `sort` function is defined in the `<algorithm>` header.
- We must use the `std::ranges::sort` namespace prefix to call the range version of the function.
- For ascending sort, you simply pass the container (`fruitsToSort`). For descending, you pass the container and the predicate (`std::greater<>{}`).

Output

```
Ranges Ascending Sort: Done.
Ranges Descending Sort: Done.
```


❖ 7.5.4. Key Points to Remember

- `ranges`**:** A range is a pair of iterators (`begin` and `end`) representing a sequence of elements.
- `std::sort`**:** Sorts elements in place. Use a predicate like `std::greater<>` for descending order.
- `std::find`**:** Finds an element in a container and returns an iterator pointing to it or the end if not found.
- `std::distance`**:** Computes the index of an element from the beginning iterator.
- **C++20 Ranges:** Simplifies algorithms by letting you pass the container directly instead of iterators.

❖ 7.5.5. Demo: Using Common Algorithms

Demo 7.5: StandardDataStructures/Demos/CommonAlgorithmDemo.cpp

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>   // sort, find, ranges::sort
#include <functional>  // std::greater

int main() {
    // --- Sort Demo ---
    std::vector<std::string> fruits = {
        "Pears", "Bananas", "Oranges", "Apples", "Lemons"
    };

    // Ascending sort
    std::sort(fruits.begin(), fruits.end());
    std::cout << "--- Ascending Sort ---\n";
    for (auto& fruit : fruits) {
        std::cout << fruit << "\n";
    }

    // Descending sort
    fruits = { "Pears", "Bananas", "Oranges", "Apples", "Lemons" };
    std::sort(fruits.begin(), fruits.end(), std::greater<>());
    std::cout << "\n--- Descending Sort ---\n";
    for (auto& fruit : fruits) {
        std::cout << fruit << "\n";
    }

    // --- Find Demo ---
    std::vector<std::string> searchFruits = {
        "Pears", "Bananas", "Oranges", "Apples", "Lemons", "Bananas"
    };

    auto it = searchFruits.begin();
    std::cout << "\n--- Find All Occurrences of 'Bananas' ---\n";

    while ((it = std::find(it, searchFruits.end(), "Bananas")) != search
         Fruits.end()) {
        std::cout << "Found at index: "
                  << std::distance(searchFruits.begin(), it) << "\n";
        ++it; // move forward to keep searching
    }

    // --- Ranges Sort Demo ---
    std::vector<std::string> moreFruits = {
```

Demo 7.5: StandardDataStructures/Demos/CommonAlgorithmDemo.cpp

```
44.           "Pears", "Bananas", "Oranges", "Apples", "Lemons"
45.       };
46.
47.       std::ranges::sort(moreFruits);
48.       std::cout << "\n--- Ranges Ascending Sort ---\n";
49.       for (auto& f : moreFruits) {
50.           std::cout << f << "\n";
51.       }
52.
53.       std::ranges::sort(moreFruits, std::greater<>());
54.       std::cout << "\n--- Ranges Descending Sort ---\n";
55.       for (auto& f : moreFruits) {
56.           std::cout << f << "\n";
57.       }
58.       return 0;
59.   }
```

Exercise 7: Explore and Implement Standard Data Structures

15 to 30 minutes

In this exercise, you will enhance the **Bank Menu** app by recording deposits and withdrawals using modern C++ data structures. You will also practice using algorithms such as `std::sort` to organize the recorded values.

❖ E7.1. Requirements

1. Use **Standard Library containers** inside the Banking Menu.
2. Convert the existing C-style arrays into modern containers.
3. Use a **vector** to store deposits.
4. Use a **deque** to store withdrawals.
5. Create a **map** to keep track of total deposited and withdrawn amounts.
6. Use the **sort algorithm** to display values in descending sequence.

❖ E7.2. Step-by-step Instructions

1. Open `StandardDataStructures\Exercises\BakingMenu.cpp` as your starter file.
2. Convert the `depositLog` array into a `std::vector<double>`.
3. Convert the `withdrawalLog` array into a `std::deque<double>`.
4. Define a map named `transactionsMap`. The key will be `std::string` and the value will be `int`.
5. Initialize the map by setting:
    A. `transactionsMap["Deposits"] = 0`
    B. `transactionsMap["Withdrawals" = 0`
6. When the user selects deposit:
    A. Convert the string amount to `double`
    B. Call `deposit(amount)`
    C. Add the value to `depositLog`

D. Add the amount to `transactionsMap["Deposits"]`

7. When the user selects withdraw:
    A. Convert the string amount to `double`
    B. Call `withdraw(amount)`
    C. Add the value to the front of `withdrawDeque`
    D. Add the amount to `transactionsMap["Withdrawals"]`
8. When the user selects quit:
    A. Print withdrawals:
        i. As a stack so most recent withdrawals are displayed first.
        ii. As a queue so least withdrawals are displayed first.
    B. Display deposits in descending order and print them
9. Set `account.balance` to the final balance.
10. Print the full account report using `printAccountStruct()`.
11. Print the sum of the deposits and the withdrawals.
12. Compile and test your program.

Solution: StandardDataStructures/Solutions/BankingMenu.cpp

```
1.  #include <iostream>
2.  #include <format>
3.  #include <stdexcept>
4.  #include <string>
5.  #include <vector>
6.  #include <deque>
7.  #include <map>
8.  #include <algorithm>
9.
10. typedef std::string str;
11.
12. const double START_BALANCE = 0.0; // Initial account balance
13. double balance = START_BALANCE;   // User's account balance
14. str userInput;
15.
16. struct Account {
17.     int acctNbr;
18.     str holder;
19.     double balance;
20.
21.     enum class AcctType {
22.         Checking,
23.         Savings
24.     };
25.
26.     AcctType acctType;
27. };
28.
29. str displayAcctType(Account::AcctType acctType) {
30.     switch (acctType) {
31.     case Account::AcctType::Checking:
32.         return "Checking";
33.     case Account::AcctType::Savings:
34.         return "Savings";
35.     default:
36.         return "Invalid Account Type";
37.     }
38. }
39.
40. void printAccountStruct(Account& account) {
41.     std::cout << std::format("{:^70}\n", "Bank Account Report");
42.     std::cout << std::format("{:<15} {:<20}\n", "Account Number", account.ac ↵↵
          ctNbr);
43.     std::cout << std::format("{:<15} {:<20}\n", "Account Holder", account.holder);
```

Solution: StandardDataStructures/Solutions/BankingMenu.cpp

```
44.      std::cout << std::format("{:<15} {:<20}\n", "Account Type", displayAcctType(ac ↵↵
             count.acctType));
45.      std::cout << std::format("{:<15} ${:<20.2f}\n", "Balance", account.balance);
46.  }
47.
48.  double withdraw(double amount) {
49.      static int nbrOfWithdrawals = 1;
50.      if (amount > balance) {
51.          std::string error_msg = std::format("Withdrawal rejected: amount ${:.2f}
                exceeds balance.\n", amount);
52.          throw std::runtime_error(error_msg);
53.      }
54.      else if (amount <= 0) {
55.          std::string error_msg = std::format("Withdrawal rejected: amount ${:.2f}
                is less than or equal to zero.\n", amount);
56.          throw std::runtime_error(error_msg);
57.      }
58.      else {
59.          std::cout << std::format("Number of withdrawals is now {}\n",
              nbrOfWithdrawals);
60.          if (nbrOfWithdrawals > 3) {
61.              amount += 3;
62.          }
63.          nbrOfWithdrawals++;
64.          return balance -= amount;
65.      }
66.  }
67.
68.  double deposit(double amount) {
69.      if (amount <= 0) {
70.          std::string error_msg = std::format("Deposit rejected: amount ${:.2f}
              is less than or equal to zero.\n", amount);
71.          throw std::runtime_error(error_msg);
72.      }
73.      else {
74.          return balance += amount;
75.      }
76.  }
77.
78.  int main() {
79.      //Allocate memory for the struct
80.      Account account;
81.
82.      str userInput;
83.      double amount;
```

Solution: StandardDataStructures/Solutions/BankingMenu.cpp

```
84.
85.       std::vector<double> depositLog{};  // use a vector for deposits
86.       std::deque<double> withdrawDeque; // use deque for withdrawals
87.       std::map<std::string, double> transactionsMap;  // use map to store deposit
                and withdrawal totals
88.       // set total deposits to 0
89.       transactionsMap["Deposits"] = 0;
90.       // set total withdrawals to 0
91.       transactionsMap["Withdrawals"] = 0;
92.
93.
94.       account.acctNbr = 555666777;
95.       // prompt user for account holder name
96.       std::cout << "Please enter account holder name: ";
97.       getline(std::cin, userInput);
98.       account.holder = userInput;
99.
100.      bool correct = false;
101.      // prompt user for account type
102.      while (!correct && userInput[0] != 'Q') {
103.          std::cout << "Enter account type (C for Checking or S for Savings): ";
104.          getline(std::cin, userInput);
105.          switch (userInput[0]) {
106.          case 'C':
107.          case 'c':
108.              account.acctType = Account::AcctType::Checking;
109.              correct = true;
110.              break;
111.          case 'S':
112.          case 's':
113.              account.acctType = Account::AcctType::Savings;
114.              correct = true;
115.              break;
116.          default:
117.             std::cout << "Unknown account type...please re-enter (Q to quit).\n";
118.          }
119.      }
120.
121.      do {
122.          // Display menu options
123.          std::cout << "\n Deposit (d)\n";
124.          std::cout << " Withdraw (w)\n";
125.          std::cout << " Balance (b)\n";
126.          std::cout << " Quit (q)\n\n";
```

Solution: StandardDataStructures/Solutions/BankingMenu.cpp

```
127.          std::cout << "Enter choice: ";
128.          std::getline(std::cin, userInput);
129.          switch (userInput[0]) {
130.              // Handle deposit
131.              case 'd':
132.              case 'D': {
133.                  double amount;
134.                  std::cout << "Enter amount to deposit: ";
135.                  std::getline(std::cin, userInput);
136.                  try {
137.                      amount = std::stod(userInput);
138.                      balance = deposit(amount);
139.                      // log deposit amount to vector
140.                      depositLog.push_back(amount);
141.                      // add deposit amount to running total on the map
142.                      transactionsMap["Deposits"] += amount;
143.                      std::cout << std::format("Deposited ${:.2f}\n", amount);
144.                  }
145.                  catch (std::invalid_argument& ie) {
146.                      std::cout <<  "Deposit amount must be numeric, amount reject ↵↵
            ed.\n";
147.                  }
148.                  catch (std::runtime_error& re) {
149.                     std::cout << re.what();
150.                  }
151.                  break;
152.              }
153.              // Handle withdrawal
154.              case 'w':
155.              case 'W': {
156.                  double amount;
157.                  std::cout << "Enter withdrawal amount: ";
158.                  std::getline(std::cin, userInput);
159.                  try {
160.                      amount = std::stod(userInput);
161.                      balance = withdraw(amount);
162.                      // log withdrawal to deque
163.                      // use push_front so that most recent withdrawal is front
            of the deque
164.                      withdrawDeque.push_front(amount);
165.                      // add withdrawal amount to running total on the map
166.                      transactionsMap["Withdrawals"] += amount;
167.                      std::cout << std::format("Withdrew ${:.2f}\n", amount);
168.                  }
```

Solution: StandardDataStructures/Solutions/BankingMenu.cpp

```
169.                 catch (std::invalid_argument& ie) {
170.                  std::cout <<  "Withdrawal amount must be numeric, amount re  ↵↵
            jected.\n";
171.                 }
172.                 catch (std::runtime_error& rte) {
173.                     std::cout << rte.what();
174.                 }
175.                 break;
176.             }
177.             // Display balance
178.             case 'b':
179.             case 'B':
180.                std::cout << std::format("Current balance: ${:.2f}\n", balance);
181.                 break;
182.             // Handle quit
183.             case 'q':
184.             case 'Q':
185.                 std::cout << std::format("Final balance: ${:.2f}\n", balance);
186.                 break;
187.             // Handle invalid input
188.             default:
189.               std::cout << "Invalid selection. Please choose a valid option.\n";
190.         }
191.
192.     } while (userInput[0] != 'q' && userInput[0] != 'Q');
193.
194.     // print withdrawDeque as a stack so more recent withdrawals are displayed
           first
195.     std::cout << std::format("{:^20}\n", "Log of Withdrawals-Most Recent First");
196.     for (auto& wd : withdrawDeque) {
197.         std::cout << std::format("{:>20}\n", std::format("${:.2f}", wd));
198.     }
199.     // print withdrawDeque as a queue so least recent withdrawals are displayed
            first
200.     std::cout << std::format("{:^20}\n", "Log of Withdrawals-Least Recent First");
201.     for (auto wdIter = withdrawDeque.rbegin(); wdIter != withdrawDeque.rend();
           wdIter++) {
202.         std::cout << std::format("{:>20}\n", std::format("${:.2f}", *wdIter));
203.     }
204.     // print deposit log
205.     std::cout << std::format("{:^20}\n", "Log of Deposits");
206.     std::sort(depositLog.begin(), depositLog.end(), std::greater<>{});  // sort
            deposits in descending sequence
207.     for (auto& deposit : depositLog) {
208.         if (deposit == 0) {
```

Solution: StandardDataStructures/Solutions/BankingMenu.cpp

```
209.             break;
210.         }
211.         std::cout << std::format("{:>20}\n", std::format("${:.2f}", deposit));
212.     }
213.     // set the balance in the account struct
214.     account.balance = balance;
215.     // print account struct
216.     printAccountStruct(account);
217.     // print totals from transactions map
218.     std::cout << std::format("Total amount of deposits is ${:.2f}.\n", transac ↵↵
             tionsMap["Deposits"]);
219.     std::cout << std::format("Total amount of withdrawals is ${:.2f}.\n",
             transactionsMap["Withdrawals"]);
220. }
```


Conclusion

In this lesson, you learned the core ideas behind commonly used data structures and how they help you store and organize data efficiently. We reviewed linear structures such as arrays and linked lists, and saw how arrays offer fast indexed access while linked lists provide flexible insertion and deletion.

You also explored stacks and queues, which show why the order of operations matters. Stacks follow a last-in, first-out approach, while queues follow first-in, first-out. These structures appear everywhere, from handling function calls to managing tasks and buffering data.

By understanding the strengths, trade-offs, and typical use-cases of each structure, you are now better equipped to choose the right tool for the job. This ability is essential for writing efficient, clear, and scalable programs.

LESSON 8

Object-Oriented Programming

Topics Covered

☑ Core OOP principles: encapsulation, inheritance, and polymorphism.

☑ Defining classes and creating objects.

☑ Declaring and using member variables and member functions.

☑ Managing the object lifecycle with constructors and destructors.

☑ Controlling visibility with access modifiers (public, private, protected).

☑ Implementing operator overloading to define custom operator behavior.

Introduction

In this lesson, you'll learn how to think in terms of objects and model simple real-world problems using C++. We'll walk through the core ideas of OOP: encapsulation with private data and getters/setters, inheritance for reusing code, and polymorphism through virtual functions and overrides. You'll also understand how access levels (public, private, protected) help keep class design clean and safe.

You will define your own classes, write member functions, and manage objects using constructors and destructors. We also introduce function and operator overloading so your classes can behave more naturally. By the end, you'll create an `Account` class and build a small banking menu program to practice designing, creating, and using objects confidently.

8.1. OOP Principles: Encapsulation, Inheritance, Polymorphism

Object Oriented Programming or **OOP** refers lets you model real-world concepts using **classes** and **objects**. A **class** is a blueprint, and an **object** is created from that blueprint.

OOP is built on three pillars:

- **Encapsulation**
- **Inheritance**
- **Polymorphism**

Encapsulation means **grouping** related data and functions together and **protecting** the internal details of an object from the outside world.

- A class can hide the implementation part and discloses only the functionalities required by other classes.
- It helps in better maintainability, readability and usability.

Inheritance lets one class **extend** another. It is the mechanism by which one class is allowed to inherit the features (fields and methods) of another class. Inheritance means creating new classes based on existing ones. We will cover inheritance in an upcoming lesson.

Polymorphism literally means having **many forms**. Different classes to provide their own form (implementation) of a function with the same name. We will cover polymorphism in an upcoming lesson.

❖ 8.1.1. Encapsulation

Encapsulation groups data and functions that are related. In addition, encapsulation controls **access** to this data.

As an example, let's consider a Person class with the following member variables:

- First Name
- Last Name

We wish to hide these variables, or attributes, so they cannot be directly read or modified by the outside world (i.e., C++ code outside of the class definition).

Here is the class definition:

```
class Person {
// Access control - private
private:
    std::string firstName;
    std::string lastName;
// Access control - public
public:
    // Constructor with first name and last name
    Person(const std::string& fName, const std::string& lName)
        : firstName(fName), lastName(lName) {
    }
    // Overloaded constructor with last name only
    Person(const std::string& lName)
        : lastName(lName) {
        setFirstName("");
    }
    // Default constructor no parameters
    Person() {
        setFirstName("");
        setLastName("");
    }

    // Getters
    std::string getFirstName() const { return firstName; }
    std::string getLastName() const { return lastName; }

    // Setters
    void setFirstName(const std::string& fName) { firstName = fName; }
    void setLastName(const std::string& lName) { lastName = lName; }

    // Destructor
    ~Person()

    // overload == operator
    bool operator==(Person& anotherPerson) {
       return (firstName == anotherPerson.firstName && lastName == anotherPerson.last ↵↵
Name);
    }
};
```


Note the following:

- `class Person` tells the compiler we are creating a class definition. The name of the class is `Person`.
- The variables `firstName` and `lastName` are declared as private. These variables **cannot be accessed** by external C++ programs.
- The constructor function is overloaded. Three forms of the constructor are present
    - `Person(const std::string& fName, const std::string& lName)` takes first name and last name as parameters.
    - `Person(const std::string& lName)` takes last name as a parameter.
    - `Person()` is the default constructor and takes 0 parameters.
- **Setter** and **getter** functions are defined to control access to each of the member variables.

We will go into more detail about this class in the next activity.

8.2. Class Definitions

In Object-Oriented Programming (OOP), a **class** serves as a blueprint or template for creating **objects** (also called instances). A class defines:

- **member variables**
- **member functions**

An **object** is created from a class. Each object gets its own copy of the member variables defined in that class.

The process of creating an object from a class is called **instantiation**.

❖ 8.2.1. Defining a Class

A class definition uses the `class` keyword followed by the class name. By convention, class names start with an uppercase letter (e.g., `Person`).

Below is a simple `Person` class containing a first name and last name. It also provides **getters** (read functions) and **setters** (update functions).

```
class Person {
private:
    std::string firstName;
    std::string lastName;

public:
    // Constructor
    Person(const std::string& fName, const std::string& lName)
        : firstName(fName), lastName(lName) {
    }

    // Getters
    std::string getFirstName() { return firstName; }
    std::string getLastName()  { return lastName; }

    // Setters
    void setFirstName(const std::string& fName) { firstName = fName; }
    void setLastName(const std::string& lName)  { lastName = lName; }
};
```

Note the following:

- A class is created using the `class` keyword and a capitalized name, such as `Person`.
- The **private** section contains variables hidden from outside the class and this enforces **encapsulation**.
- The **public** section contains functions that outside code is permitted to call.
- The **constructor** is a special function that initializes the object. Its name is always the same as the class (`Person`).
- The constructor uses an **initializer list**, shown after the colon (`:`).
- Lines like `firstName(fName)` set the private member variable directly.
- Parameters such as `const std::string& fName` use **pass-by-reference** to avoid expensive string copies.
- The `const` keyword is used to prevent the constructor from altering the value of each parameter.
- **Getters** are functions that return the value of a private member variable. By convention we prepend `get` to the variable name (and uppercase the first letter of the variable name) to form the name of the getter function.
    - The return type of the function is the data type of the variable
    - The variable is returned to the caller using the `return` statement.

- **Setters** are functions that set or modify the value of a private member variable. By convention we prepend `set` to the variable name (and uppercase the first letter of the variable name) to form the name of the setter function.
    - The return type of the function is `void`.
    - The variable is updated by assigning the value of the variable to the parameter passed to the setter.
- If a parameter has the same name as the member variable, use `this->`:

```
void setLastName(const std::string& lastName) {
    this->lastName = lastName;
}
```

We use the `this->` notation to differentiate the member variable name from the parameter name. The reserved keyword `this` contains the address of the object. This prevents name confusion. `this` refers to the **current object**.

❖ 8.2.2. Getters and Setters

Getters and Setters are public member functions that control access to the private member variables, maintaining encapsulation.

Example: Defining Getters and Setters

```
class Person {
// private members
public:
    // constructor

    // Getters: Return the value of a private member
    std::string getFirstName() const { return firstName; }

    // Setters: Modify the value of a private member
    void setLastName(const std::string& lName) { lastName = lName; }

    // Alternative Setter syntax using 'this'
    void setFirstName(const std::string& firstName) {
        this->firstName = firstName;
    }
};
```

Note the following:

- Getters are typically marked `const` (e.g., `getFirstName() const`) to tell the compiler they will not modify the object's state.
- Setters usually have a `void` return type and take the new value as a parameter.
- The keyword `this` is a reserved pointer that holds the memory address of the current object. `this->firstName` is used to explicitly distinguish the member variable from the parameter when they share the same name.

Note: You will learn more about getters, setters and constructors in later lessons.

❖ 8.2.3. Advantages of Using Classes

Classes provide:

- **Encapsulation:** Data is protected using private variables.
- **A reusable structure:** You create many objects from one blueprint.
- **Better code organization:** Related data and functionality stay together.
- **Clear responsibilities:** Each class handles its own data and behavior.

❖ 8.2.4. Using a Class

Let's create a `Person` object and use its getters.

```
// create a Person
Person person("Joe", "Smith");

std::cout << "The person is "
          << person.getFirstName() << " "
          << person.getLastName() << ".\n";
```

Note the following:

- `Person person("Joe", "Smith");` calls the constructor and initializes the object.
- The object `person` now stores first and last name privately.
- We use `person.getFirstName()` and `person.getLastName()` to access private data safely.
- Direct access such as `person.firstName` is not allowed, which protects data integrity.

Here is the output:

```
The person is Joe Smith.
```


8.3. Member Variables and Functions

A class is a blueprint for how an object should **look and behave**. That blueprint is made of two key things:

- **Member variables** means the data
- **Member functions** means the behavior

Together, they allow a class to hold information and control how that information is accessed or modified.

❖ 8.3.1. Member Variables

Member **variables** (sometimes called attributes or fields) store the data for each object created from a class. Every object created from the class gets its own copy of these variables.

Member variables:

- represent the **state** of an object.
- describe the **properties** of the object.
- are usually kept **private** to protect the data.

In C++, member variables are typically declared in the `private` section of the class. This enforces **encapsulation**, meaning the data is hidden and protected from direct manipulation by external code. Therefore access to the member variables is controlled by the function logic..

```
class Person {
private:
    std::string firstName;
    std::string lastName;
```

Note the following:

`private:` declares the beginning of **private variables**, i.e., variables that can only be accessed within the class.

The member variables are `firstName` and `lastName`.

❖ 8.3.2. Member Functions

Member **functions**, sometimes called **methods**, are functions defined within a class to implement the behavior and actions that an object can perform. They can manipulate member variables and provide functionality to the objects.

Member functions define the object's behavior. Typical tasks of member functions:

- read data
- update data
- perform actions

Member functions provide the public interface for the object. We've previously discussed three key types:

- **Constructors:** Special functions used to initialize the object when it is created.
- **Getters:** Functions used to safely read (get) the value of a private member variable.
- **Setters:** Functions used to safely modify (set) the value of a private member variable, often including validation rules.

We return to our `Person` class example:

```
// Constructor with first name and last name
   Person(const std::string& fName, const std::string& lName)
       : firstName(fName), lastName(lName) {
   }
...
// Getters
   std::string getFirstName() const { return firstName; }
...
   // Setters
   void setFirstName(const std::string& fName) { firstName = fName; }
```

Items of interest:

- The constructor function must have the same name as the class, in this case `Person`.
- The `getFirstName` function allows external code to **read** the private `firstName` value.
- The `setFirstName` function allows external code to **modify** the private `lastName` value.
- Setters and getters implement **encapsulation** (data hiding) of private member variables.

Use the following file to demonstrate constructor, getter, and setter functions.

Demo 8.1: ObjectorientedProgramming/Demos/ObjectOrientedProgramming.cpp

```
1.   #include <iostream>
2.   #include <format>
3.
4.   class Person {
5.   // Access control - private
6.   private:
7.       std::string firstName;
8.       std::string lastName;
9.   // Access control - public
10.  public:
11.      // Constructor with first name and last name
12.      Person(const std::string& fName, const std::string& lName)
13.          : firstName(fName), lastName(lName) {
14.      }
15.      // Overloaded constructor with last name only
16.      Person(const std::string& lName)
17.          :lastName(lName) {
18.          setFirstName("");
19.      }
20.      // Default constructor no parameters
21.      Person() {
22.          setFirstName("");
23.          setLastName("");
24.      }
25.
26.      // Getters
27.      std::string getFirstName() const { return firstName; }
28.      std::string getLastName() const { return lastName; }
29.
30.      // Setters
31.      void setFirstName(const std::string& fName) { firstName = fName; }
32.      void setLastName(const std::string& lName) { lastName = lName; }
33.
34.      // Destructor
35.      ~Person() { }
36.
37.      // overload == operator
38.      bool operator==(Person& anotherPerson) {
39.         return (firstName == anotherPerson.firstName && lastName == anotherPer  ↵↵
             son.lastName);
40.      }
41.  };
42.
```

Demo 8.1: ObjectorientedProgramming/Demos/ObjectOrientedProgramming.cpp

```
43.  int main() {
44.      // create a Person with first name and last name
45.      Person person("Joe", "Smith");
46.      std::cout << std::format("The person is {} {}.\n", person.getFirstName(),
             person.getLastName());
47.      // create a Person using last name constructor
48.      Person person2("Gilmour");
49.      person2.setFirstName("David");   // call setter to set first name
50.      std::cout << std::format("The person created with last name constructor is
             {} {}.\n", person2.getFirstName(), person2.getLastName());
51.      // create a person using default constructor
52.      Person person3;
53.      person3.setFirstName("Misty");   // call setter to set first name
54.      person3.setLastName("Autumn");    // call setter to set last name
55.      std::cout << std::format("The person created with default constructor is {}
             {}.\n", person3.getFirstName(), person3.getLastName());
56.      // create a duplicate person
57.      Person duplicatePerson("Joe", "Smith");
58.      std::cout << "\n==== Duplicate Person check ====\n";
59.      if (person == duplicatePerson) {
60.          std::cout << "Persons are the same!\n";
61.      }
62.      else {
63.          std::cout << "Persons are NOT the same!\n";
64.      }
65.  }
```


8.4. Constructors and Destructors

When you want to create an object in C++, a **constructor** function runs automatically when the object is created and a **destructor** runs automatically when the object is destroyed.

These two help you control the object's lifetime in a safe and predictable way.

❖ 8.4.1. Constructors

A **constructor** is a special member function that is automatically called when an object is **instantiated** (created).

Important rules:

- A constructor is a member function of a class that has the **same name** as the class name.
- It helps to **initialize** the object of a class.
- It has **no return type**.
- It is called **automatically** when an object is created.
- It can be defined manually with arguments or without arguments.

Types of Constructors

- **Default Constructor** (No-Argument Constructor): A constructor that takes no parameters. If you don't write any constructors yourself, the compiler will automatically generate a public default constructor for you.
- **Parameterized Constructor:** A constructor that accepts one or more arguments to set the initial state of the object.

Syntax

```
class className {

    // Constructor
    className() {
        // Body of constructor
    }
};
```

Example: A Simple Constructor

```
#include <iostream>
#include <string>

class Person {
private:
    std::string firstName{};
    std::string lastName{};

public:
    // Constructor
    Person(const std::string& fName, const std::string& lName)
        : firstName(fName), lastName(lName) {
    }
};
int main() {
    Person person("Joe", "Smith");
    std::cout << "Person object created successfully!\n";
}
```


Note the following:

- The constructor has the same name as the class: `Person`.
- No return type is coded.
- Members are initialized using an **initializer list**, a modern and recommended practice.
- `const std::string&` avoids copying of the string because a **reference** is passed to the constructor. `const` ensures the constructor cannot modify the parameter.

Output

```
Person object created successfully!
```

❖ 8.4.2. Destructors

The **destructor** is the counterpart to the constructor. It is a special member function that is automatically called when an object is about to be **destroyed** (i.e., when it goes out of scope or is explicitly deleted).

Important rules:

- The destructor name is the class name prepended with a tilde (~), for example, `~Person()`.

- A class can only have one destructor, and it takes **no parameters**.
- It helps to **deallocate** the memory of an object.
- It is called while the object of the class is **freed** or **deleted**.
- Called automatically when:
    - The object goes out of scope
    - Is explicitly deleted (for objects created using `new`).
- If you do not write a destructor, the compiler **creates one** automatically.
- If a class is inherited by another class and both the classes have a destructor then the destructor of the child class is called first, followed by the destructor of the parent or base class.

Syntax

```
class className {

    // Destructor
    ~className() {
        // Body of destructor
    }
};
```


Example: Destructor Running Automatically

```
#include <iostream>

class Demo {
public:
    Demo() {
        std::cout << "Constructor called.\n";
    }
    ~Demo() {
        std::cout << "Destructor called.\n";
    }
};
int main() {
    Demo d;
}
```

Note the following:

- The destructor name is `~Demo`.

- It runs after main() finishes, just before the object is destroyed.
- No parentheses or parameters can be added.

Output

```
Constructor called.
Destructor called.
```

When Do You need a Custom Destructor?

A destructor is crucial when you use the `new` keyword to allocate memory for things like arrays or objects. You only need to write your own destructor if your class manually allocates memory or resources, such as:

- memory using `new`
- file handles
- network connections
- database connections

If you do not allocate resources manually, you usually do not need a destructor.

Example: Cleaning Up Dynamic Memory

```
#include <iostream>

class MyClass {
private:
    int* data;

public:
    MyClass() {
        data = new int[5]; // Allocate memory
        std::cout << "Memory allocated.\n";
    }
    ~MyClass() {
        delete[] data; // Free memory
        std::cout << "Memory freed.\n";
    }
};
int main() {
    MyClass obj;
}
```

Note the following:

- `new[]` allocates an array on the heap.
- `delete[]` deallocates the memory.
- Destructors prevent memory leaks.
- This is one main reason destructors exist.

Output

```
Memory allocated.
Memory freed.
```

❖ 8.4.3. Difference between Constructor and Destructor

Constructor	**Destructor**
Constructor helps to initialize the object of a class.	Destructor is used to destroy the instances.

It is declared as:
`className( arguments if any )`
`{Constructor's Body }`
	It is declared as:
	`~ className( no arguments )`
	`{ }`

Constructor can either accept arguments or not.	It can't have any arguments.
A constructor is called when an instance or object of a class is created.	It is called while object of the class is freed or deleted.
Constructor is used to allocate the memory to an instance or object.	It is used to deallocate the memory of an object of a class.
Constructor can be overloaded.	It can't be overloaded.
The constructor's name is same as the class name.	Its name is also same as the class name preceded by the tiled (~) operator.
In a class, there can be multiple constructors.	In a class, there is always a single destructor.
There is a concept of copy constructor which is used to initialize an object from another object.	There is no copy destructor concept.
They are often called in successive order.	They are often called in reverse order of constructor.

❖ 8.4.4. Demo: Constructors and Destructors

This demo shows the exact points in the program execution where the constructor and destructor are automatically called.

Demo 8.2: ObjectorientedProgramming/Demos/Constructors.cpp

```
1.  #include <iostream>
2.
3.  class MyClass {
4.  private:
5.      int* data;
6.
7.  public:
8.      MyClass() {
9.          data = new int[5]; // Allocate memory
10.         std::cout << "Memory allocated.\n";
11.     }
12.     ~MyClass() {
13.         delete[] data; // Free memory
14.         std::cout << "Memory freed.\n";
15.     }
16. };
17. int main() {
18.     MyClass obj;
19. }
```


8.5. Access Control

Access control protects data inside a class by controlling how the data is **accessed** or **modified**.

This is an important part of **encapsulation**, one of the core ideas of object-oriented programming. The data is hidden so that the only way outside programs can access the data is through member functions.

❖ 8.5.1. `private` Access

Private access makes class members accessible only inside the class itself.

`private` is the strictest access level. By default, all class members in C++ are private if no specifier is mentioned.

Important rules:

- Can only be accessed **only inside the class**.
- **Not accessible** from outside code.
- Protects data by controlling access and modification through functions.

❖ 8.5.2. `public` Access

Public access makes class members (variables and functions) accessible from outside the class.

`public` is the most permissive access level. It is commonly used for functions that external programs can call.

❖ 8.5.3. `protected` Access

`protected` offers an intermediate level of access, designed specifically for use with **inheritance**.

- The protected specifier makes members accessible inside the class itself and in its **derived** (child) classes, but not from the outside code.
- It is mainly used in inheritance, allowing child classes to reuse or modify data and functions while still keeping them hidden from the outside world.

We will look at inheritance and protected access in more detail in an upcoming lesson.

Examples

The following example shows a constructor, a getter and a setter function that are each declared as public:

```
class Person {
...
// Access control - public
public:
    // Constructor with first name and last name
    Person(const std::string& fName, const std::string& lName)
        : firstName(fName), lastName(lName) {
    }
...
    // Getter
    std::string getFirstName() const { return firstName; }
...
    // Setter
    void setFirstName(const std::string& fName) { firstName = fName; }

...
```


This example shows the private member variables of the Person class:

```
class Person {
// Access control - private
private:
    std::string firstName;
    std::string lastName;
...
```

You can find these examples in the following demo file.

Demo 8.3: ObjectorientedProgramming/Demos/ObjectOrientedProgramming.cpp

```
1.   #include <iostream>
2.   #include <format>
3.
4.   class Person {
5.   // Access control - private
6.   private:
7.       std::string firstName;
8.       std::string lastName;
9.   // Access control - public
10.  public:
11.      // Constructor with first name and last name
12.      Person(const std::string& fName, const std::string& lName)
13.          : firstName(fName), lastName(lName) {
14.      }
15.      // Overloaded constructor with last name only
16.      Person(const std::string& lName)
17.          :lastName(lName) {
18.          setFirstName("");
19.      }
20.      // Default constructor no parameters
21.      Person() {
22.          setFirstName("");
23.          setLastName("");
24.      }
25.
26.      // Getters
27.      std::string getFirstName() const { return firstName; }
28.      std::string getLastName() const { return lastName; }
29.
30.      // Setters
31.      void setFirstName(const std::string& fName) { firstName = fName; }
32.      void setLastName(const std::string& lName) { lastName = lName; }
33.
34.      // Destructor
35.      ~Person() { }
36.
37.      // overload == operator
38.      bool operator==(Person& anotherPerson) {
39.         return (firstName == anotherPerson.firstName && lastName == anotherPer ↵↵
             son.lastName);
40.      }
41.  };
42.
```

Demo 8.3: ObjectorientedProgramming/Demos/ObjectOrientedProgramming.cpp

```
43.  int main() {
44.      // create a Person with first name and last name
45.      Person person("Joe", "Smith");
46.      std::cout << std::format("The person is {} {}.\n", person.getFirstName(),
             person.getLastName());
47.      // create a Person using last name constructor
48.      Person person2("Gilmour");
49.      person2.setFirstName("David");   // call setter to set first name
50.      std::cout << std::format("The person created with last name constructor is
             {} {}.\n", person2.getFirstName(), person2.getLastName());
51.      // create a person using default constructor
52.      Person person3;
53.      person3.setFirstName("Misty");   // call setter to set first name
54.      person3.setLastName("Autumn");    // call setter to set last name
55.      std::cout << std::format("The person created with default constructor is {}
             {}.\n", person3.getFirstName(), person3.getLastName());
56.      // create a duplicate person
57.      Person duplicatePerson("Joe", "Smith");
58.      std::cout << "\n==== Duplicate Person check ====\n";
59.      if (person == duplicatePerson) {
60.          std::cout << "Persons are the same!\n";
61.      }
62.      else {
63.          std::cout << "Persons are NOT the same!\n";
64.      }
65.  }
```


8.6. Function and Operator Overloading

C++ allows you to use the same function name for multiple different purposes, as long as the compiler can tell them apart. This ability is called **overloading**.

It is a form of polymorphism in C++ that enables a single function name or operator symbol to have **many forms** (behaviors) depending on the number or type of operands/arguments used.

There are two kinds:

1. **Function overloading**
2. **Operator overloading**

❖ 8.6.1. Function Overloading

Function overloading allows you to create **multiple functions** with the **same name**, but each version must have different parameter types or a different number of parameters.

The compiler decides which function to call based on:

- the number of parameters
- the types of those parameters

Overloading Constructors

Constructors can be overloaded too. Constructors are the most common functions to be overloaded, allowing an object to be initialized in several different ways.

Below, the `Person` class has two constructors:

1. The first constructor takes first name and last name
2. The second constructor takes only a last name

Example

```
// Constructor with first name and last name
Person(const std::string& fName, const std::string& lName)
    : firstName(fName), lastName(lName) {
}

// Overloaded constructor with last name only
Person(const std::string& lName)
    : lastName(lName) {
    setFirstName("");
}
```

Items of Interest

- Both constructors have the same name (`Person`) because constructors must match the class name.
- The compiler distinguishes the constructors because:

- The first constructor has **two** parameters
- The second constructor has **one** parameter

- In the second constructor we initialize `firstName` to an empty string (`""`) to ensure the object is in a valid state, even if the user didn't provide a first name.

Testing the Overloaded Constructors

```
Person person("Joe", "Smith");
std::cout << "The person is "
          << person.getFirstName() << " "
          << person.getLastName() << "\n";

Person person2("Gilmour");
std::cout << "The person with last name only is "
          << person2.getFirstName() << " "
          << person2.getLastName() << "\n";
```

Note the following

- When creating `person`, the compiler matches the call to the constructor with two string arguments.
- When creating `person2`, the compiler matches the call to the constructor with one string argument.

Output

```
The person is Joe Smith
The person with last name only is  Gilmour
```

❖ 8.6.2. Operator Overloading

Operator overloading allows you to change the behavior of standard C++ operators such as `+`, `-`, `==`, and `!=`. Operator overloading lets you change how an operator works for your own custom types (objects).

The C++ compiler already knows how to compare integers or strings using `==`. But if you want to compare two `Person` objects, the compiler needs instructions on what "equal" means in that context.

Therefore, if we want to compare two `Person` objects, we can write:

```
if (person1 == person2) {
  ...
}
```

Overloading the == Operator

We want two `Person` objects to be considered equal if both the first name and the last names are the same:

```
class Person {
    // ...
    bool operator==(Person& anotherPerson) {
        return (firstName == anotherPerson.firstName &&
                lastName == anotherPerson.lastName);
    }
};
```

Note the following:

- The function name is the operator keyword `operator` followed by the symbol being overloaded (`==`).
- The function return type is `bool`.
- The function returns `true` if both names match.
- The function returns `false` if both names are not same.
- `firstName == anotherPerson.firstName`: This compares the `firstName` of the current object with the `firstName` of the object passed as the argument (`anotherPerson`).

Using the overloaded == operator

```
Person person("Joe", "Smith");
Person duplicatePerson("Joe", "Smith");

if (person == duplicatePerson) {
    std::cout << "Persons are the same!\n";
}
else {
    std::cout << "Persons are NOT the same!\n";
}
```

Output

```
Persons are the same!
```

❖ 8.6.3.

❖ 8.6.4. Differences between Function and Operator Overloading

Feature	**Function Overloading**	**Operator Overloading**
Meaning	Same function name, different parameters	Change behavior of operators for objects
Purpose	Allows multiple functions with the same name	Allows operators like `+`, `==`, `-` to work on objects

Example	`void print(int x);`
	
	`void print(std::string s);`
		`bool operator==(Person& p);`

Return type	Can be different	Must match logical operator behavior (e.g., `bool` for `==`)
Parameters	Must differ (number or type)	Usually one parameter (for binary operators)
Usage	Call function with different arguments	Use operator directly on objects
Flexibility	Can create many variations of a function	Can make operators work naturally for custom classes
When to use	When same action can take different inputs	When operators need custom behavior for objects
Example in class	Overloaded constructor	Overloaded `==` operator

You can find the examples in this activity in the following demo file.

Demo 8.4: ObjectorientedProgramming/Demos/ObjectOrientedProgramming.cpp

```
1.   #include <iostream>
2.   #include <format>
3.
4.   class Person {
5.   // Access control - private
6.   private:
7.       std::string firstName;
8.       std::string lastName;
9.   // Access control - public
10.  public:
11.      // Constructor with first name and last name
12.      Person(const std::string& fName, const std::string& lName)
13.          : firstName(fName), lastName(lName) {
14.      }
15.      // Overloaded constructor with last name only
16.      Person(const std::string& lName)
17.          :lastName(lName) {
18.          setFirstName("");
19.      }
20.      // Default constructor no parameters
21.      Person() {
22.          setFirstName("");
23.          setLastName("");
24.      }
25.
26.      // Getters
27.      std::string getFirstName() const { return firstName; }
28.      std::string getLastName() const { return lastName; }
29.
30.      // Setters
31.      void setFirstName(const std::string& fName) { firstName = fName; }
32.      void setLastName(const std::string& lName) { lastName = lName; }
33.
34.      // Destructor
35.      ~Person() { }
36.
37.      // overload == operator
38.      bool operator==(Person& anotherPerson) {
39.         return (firstName == anotherPerson.firstName && lastName == anotherPer ↵↵
             son.lastName);
40.      }
41.  };
42.
```

Demo 8.4: ObjectorientedProgramming/Demos/ObjectOrientedProgramming.cpp

```
43.  int main() {
44.      // create a Person with first name and last name
45.      Person person("Joe", "Smith");
46.      std::cout << std::format("The person is {} {}.\n", person.getFirstName(),
             person.getLastName());
47.      // create a Person using last name constructor
48.      Person person2("Gilmour");
49.      person2.setFirstName("David");   // call setter to set first name
50.      std::cout << std::format("The person created with last name constructor is
             {} {}.\n", person2.getFirstName(), person2.getLastName());
51.      // create a person using default constructor
52.      Person person3;
53.      person3.setFirstName("Misty");   // call setter to set first name
54.      person3.setLastName("Autumn");   // call setter to set last name
55.      std::cout << std::format("The person created with default constructor is {}
             {}.\n", person3.getFirstName(), person3.getLastName());
56.      // create a duplicate person
57.      Person duplicatePerson("Joe", "Smith");
58.      std::cout << "\n==== Duplicate Person check ====\n";
59.      if (person == duplicatePerson) {
60.          std::cout << "Persons are the same!\n";
61.      }
62.      else {
63.          std::cout << "Persons are NOT the same!\n";
64.      }
65.  }
```

Exercise 8: Designing a Basic OOP System

15 to 30 minutes

In this exercise you will create a basic `Account` class in the Banking Menu app. You will instantiate an `Account` object with the data entered by the user.

❖ E8.1. Requirements

1. Replace the `Account` struct with an `Account` class in the Banking Menu app.
2. Create two constructors for the `Account` class:
    A. `Account(int acctNbr, const std::string& holder, double balance, AcctType acctType)` , a full constructor.
    B. `Account()`, a default constructor with values:
        i. `acctNbr = 999999999`
        ii. `holder = "N/A"`
        iii. `balance = 0.0`

3. Provide getters and setters for all private members.
4. Provide a `display()` member that returns a `std::string` containing account data.
5. Create an `Account` object using the default constructor, then set `acctNbr` to `555666777` via setter.
6. Prompt the user for the account holder name and set it via setter.
7. Prompt the user for the account type: `C` = Checking, `S` = Savings, `Q` = Quit (if `Q`, exit program).
8. Present a banking menu in a `do-while` loop to accept: Deposit, Withdraw, Balance, Quit.
9. After the user exits, display account data (using the `display()` member).
10. Create an account based on user input.

❖ E8.2. Step-by-step Instructions

1. Open `ObjectOrientedProgramming\Exercises\BankingMenu.cpp` as your starter file.

2. Remove or comment out the `Account` struct.
3. After headers and `typedef` statements, add the `Account` class with the `AcctType` enum inside `public`.
4. The class requires the account type `enum` previously stored in the `struct`. Define the `enum` as shown below:

```
class Account {
public:
    enum class AcctType {
        Checking,
        Savings
    };
private:
// define acctNbr, holder, balance, and acctType here
public:
// define member functions here starting with the constructor
```

5. Define the following private member variables:

`int acctNbr;`	Account number
`std::string holder`	Name of the account holder
`double balance`	Account balance
`AcctType acctType;`	Account type

6. Create two constructors. One should accept all member variables and the other constructor will accept 0 parameters, i.e., this constructor will serve as the default constructor. Use the following values for member variables in the default constructor:
    - account number = 999999999
    - account holder = "N/A"
    - balance = 0.0
7. Write getter and setter functions for the private member variables. Mark getters `const`. Use `const&` for string parameters.
8. Create a function named `display` that returns a string containing the account data.
9. In `main()`:
    A. Create `Account account;` using default constructor.

B. Call `account.setAcctNbr(555666777);`

C. Prompt the user for the account holder. Use a setter function to update `account` with the data.

D. Prompt the user for account type using a do-while loop that validates `C`, `S`, or `Q` to quit the app.

E. lid account type is entered, inform the user that the account type is invalid and re-prompt the user.

F. If the user entered `Q` (quit) then exit the Banking menu app using `return 0;`.

10. Display the banking menu options using a do-while loop and respond to user input.
11. After the user exits the loop, call `account.display()` to show final data.
12. Create `account` as an object of type `Account` using the default constructor. Use your setter to set the account number to `555666777`.
13. Prompt the user for the account holder. Use a setter function to update `account` with the data.
14. Prompt the user for the account type using a while loop:

    A. The account type must be either `C` for Checking, `S` for Savings, or `Q` to quit the app.

    B. If an invalid account type is entered, inform the user that the account type is invalid and re-prompt the user.

    C. If the user entered `Q` (quit) then exit the Banking menu app using `return 0;`.

15. Display the banking menu options using a do...while loop and respond to user input.
16. After the user exits the loop, display the object data using the object's display function.

Solution: ObjectorientedProgramming/Solutions/BankingMenu.cpp

```
1.  #include <iostream>
2.  #include <format>
3.  #include <stdexcept>
4.  #include <string>
5.  #include <vector>
6.  #include <deque>
7.  #include <map>
8.  #include <algorithm>
9.
10. typedef std::string str;
11.
12. const double START_BALANCE = 0.0; // Initial account balance
13. str userInput;
14.
15. class Account {
16. public:
17.     enum class AcctType {
18.         Checking,
19.         Savings
20.     };
21.
22. private:
23.     int acctNbr;
24.     str holder;
25.     double balance;
26.     AcctType acctType;
27.
28. public:
29.     // Constructor with all variables
30.     Account(int acctNbr, str holder, int balance, AcctType acctType)
31.        : acctNbr(acctNbr), holder(holder), balance(balance), acctType(acctType)
              {
32.     }
33.     // default constructor (no parameters)
34.     Account() {
35.         acctNbr = 999999999;
36.         holder = "N/A";
37.         balance = 0.0;
38.     }
39.     // Getters
40.     int getAcctNbr()  { return acctNbr; }
41.     str getHolder()  { return holder; }
42.     double getBalance() { return balance; }
43.     AcctType getAcctType() { return acctType; }
```

Solution: ObjectorientedProgramming/Solutions/BankingMenu.cpp

```
44.
45.      // Setters
46.      void setAcctNbr(int acctNbr) { this->acctNbr = acctNbr; }
47.      void setHolder(str holder) { this->holder = holder; }
48.      void setBalance(double balance) { this->balance = balance; }
49.      void setAcctType(AcctType acctType) { this->acctType = acctType; }
50.
51.  };
52.
53.  str displayAcctType(Account::AcctType acctType) {
54.      switch (acctType) {
55.      case Account::AcctType::Checking:
56.          return "Checking";
57.      case Account::AcctType::Savings:
58.          return "Savings";
59.      default:
60.          return "Invalid Account Type";
61.      }
62.  }
63.
64.  void printAccountObject(Account account) {
65.      std::cout << std::format("{:^70}\n", "Bank Account Report");
66.      std::cout << std::format("{:<15} {:<20}\n", "Account Number", account.getAc ↵↵
             ctNbr());
67.      std::cout << std::format("{:<15} {:<20}\n", "Account Holder", account.getH ↵↵
             older());
68.      std::cout << std::format("{:<15} {:<20}\n", "Account Type", displayAcctType(ac ↵↵
             count.getAcctType()));
69.      std::cout << std::format("{:<15} ${:<20.2f}\n", "Balance", account.getBal ↵↵
             ance());
70.  }
71.
72.  double withdraw(double amount, double balance) {
73.      static int nbrOfWithdrawals = 1;
74.      if (amount > balance) {
75.          std::string error_msg = std::format("Withdrawal rejected: amount ${:.2f}
               exceeds balance.\n", amount);
76.          throw std::runtime_error(error_msg);
77.      }
78.      else if (amount <= 0) {
79.          std::string error_msg = std::format("Withdrawal rejected: amount ${:.2f}
               is less than or equal to zero.\n", amount);
80.          throw std::runtime_error(error_msg);
81.      }
82.      else {
```

Solution: ObjectorientedProgramming/Solutions/BankingMenu.cpp

```
83.           std::cout << std::format("Number of withdrawals is now {}\n",
               nbrOfWithdrawals);
84.           if (nbrOfWithdrawals > 3) {
85.               amount += 3;
86.           }
87.           nbrOfWithdrawals++;
88.           return balance -= amount;
89.       }
90.   }
91.
92.   double deposit(double amount, double balance) {
93.       if (amount <= 0) {
94.           std::string error_msg = std::format("Deposit rejected: amount ${:.2f}
               is less than or equal to zero.\n", amount);
95.           throw std::runtime_error(error_msg);
96.       }
97.       else {
98.           return balance += amount;
99.       }
100.  }
101.
102.  int main() {
103.
104.      str userInput;
105.
106.      std::vector<double> depositLog{};  // use a vector for deposits
107.      std::deque<double> withdrawDeque; // use deque for withdrawals
108.      std::map<std::string, double> transactionsMap;
109.      // set total deposits to 0
110.      transactionsMap["Deposits"] = 0;
111.      // set total withdrawals to 0
112.      transactionsMap["Withdrawals"] = 0;
113.
114.      // Call default constructor and use setter to set account number:
115.      Account account = Account();
116.      account.setAcctNbr(555666777);
117.      // prompt user for account holder name:
118.      std::cout << "Please enter account holder name: ";
119.      getline(std::cin, userInput);
120.      account.setHolder(userInput);
121.
122.      bool correct = false;
123.      // prompt user for account type:
124.      while (!correct && userInput[0] != 'Q') {
```

Solution: ObjectorientedProgramming/Solutions/BankingMenu.cpp

```
125.          std::cout << "Enter account type (C for Checking, S for Savings) or Q
               to Quit the app: ";
126.          getline(std::cin, userInput);
127.          switch (userInput[0]) {
128.          case 'C':
129.          case 'c':
130.              account.setAcctType(Account::AcctType::Checking);
131.              correct = true;
132.              break;
133.          case 'S':
134.          case 's':
135.              account.setAcctType(Account::AcctType::Savings);
136.              correct = true;
137.              break;
138.          case 'Q':
139.          case 'q':
140.              std::cout << "Closing the app by user request.\n";
141.              return 0;
142.          default:
143.             std::cout << "Unknown account type...please re-enter (Q to quit).\n";
144.          }
145.      }
146.
147.      do {
148.          // Display menu options
149.          std::cout << "\n Deposit (d)\n";
150.          std::cout << " Withdraw (w)\n";
151.          std::cout << " Balance (b)\n";
152.          std::cout << " Quit (q)\n\n";
153.          std::cout << "Enter choice: ";
154.          std::getline(std::cin, userInput);
155.          switch (userInput[0]) {
156.              // Handle deposit
157.              case 'd':
158.              case 'D': {
159.                  double amount;
160.                  std::cout << "Enter amount to deposit: ";
161.                  std::getline(std::cin, userInput);
162.                  try {
163.                      amount = std::stod(userInput);
164.                      account.setBalance(deposit(amount, account.getBalance()));
165.                      // log deposit amount to vector
166.                      depositLog.push_back(amount);
167.                      // add deposit amount to running total on the map
```

Solution: ObjectorientedProgramming/Solutions/BankingMenu.cpp

```
168.                     transactionsMap["Deposits"] += amount;
169.                     std::cout << std::format("Deposited ${:.2f}\n", amount);
170.                 }
171.                 catch (std::invalid_argument& ie) {
172.                  std::cout <<  "Deposit amount must be numeric, amount reject  ↵↵
            ed.\n";
173.                 }
174.                 catch (std::runtime_error& re) {
175.                    std::cout << re.what();
176.                 }
177.                 break;
178.             }
179.             // Handle withdrawal
180.             case 'w':
181.             case 'W': {
182.                 double amount;
183.                 std::cout << "Enter withdrawal amount: ";
184.                 std::getline(std::cin, userInput);
185.                 try {
186.                     amount = std::stod(userInput);
187.                     account.setBalance(withdraw(amount, account.getBalance()));
188.                     // log withdrawal to deque
189.                     // use push_front so that most recent withdrawal is front
            of the deque
190.                     withdrawDeque.push_front(amount);
191.                     // add withdrawal amount to running total on the map
192.                     transactionsMap["Withdrawals"] += amount;
193.                     std::cout << std::format("Withdrew ${:.2f}\n", amount);
194.                 }
195.                 catch (std::invalid_argument& ie) {
196.                  std::cout <<  "Withdrawal amount must be numeric, amount re  ↵↵
            jected.\n";
197.                 }
198.                 catch (std::runtime_error& rte) {
199.                     std::cout << rte.what();
200.                 }
201.                 break;
202.             }
203.             // Display balance
204.             case 'b':
205.             case 'B':
206.              std::cout << std::format("Current balance: ${:.2f}\n", account.get  ↵↵
            Balance());
207.                 break;
208.             // Handle quit
```

Solution: ObjectorientedProgramming/Solutions/BankingMenu.cpp

```
209.             case 'q':
210.             case 'Q':
211.               std::cout << std::format("Final balance: ${:.2f}\n", account.get  ↵↵
             Balance());
212.                 break;
213.             // Handle invalid input
214.             default:
215.               std::cout << "Invalid selection. Please choose a valid option.\n";
216.         }
217.     } while (userInput[0] != 'q' && userInput[0] != 'Q');
218.
219.     // print withdrawDeque as a stack so more recent withdrawals are displayed
           first
220.     std::cout << std::format("{:^20}\n", "Log of Withdrawals-Most Recent First");
221.     for (const auto& wd : withdrawDeque) {
222.         std::cout << std::format("{:>20}\n", std::format("${:.2f}", wd));
223.     }
224.     // print withdrawDeque as a queue so least recent withdrawals are displayed
            first
225.     std::cout << std::format("{:^20}\n", "Log of Withdrawals-Least Recent First");
226.     for (auto wdIter = withdrawDeque.rbegin(); wdIter != withdrawDeque.rend();
           wdIter++) {
227.         std::cout << std::format("{:>20}\n", std::format("${:.2f}", *wdIter));
228.     }
229.     // print deposit log
230.     std::cout << std::format("{:^20}\n", "Log of Deposits");
231.     std::sort(depositLog.begin(), depositLog.end(), std::greater<>{});  // sort
            depositss in descending sequence
232.     for (auto& deposit : depositLog) {
233.         if (deposit == 0) {
234.             break;
235.         }
236.         std::cout << std::format("{:>20}\n", std::format("${:.2f}", deposit));
237.     }
238.     // print account object
239.     printAccountObject(account);
240.     // print totals from transactions map
241.     std::cout << std::format("Total amount of deposits is ${:.2f}.\n", transac  ↵↵
           tionsMap["Deposits"]);
242.     std::cout << std::format("Total amount of withdrawals is ${:.2f}.\n",
           transactionsMap["Withdrawals"]);
243. }
```

Conclusion

In this lesson, you explored the core foundations of Object-Oriented Programming by defining classes and objects, using member variables and methods, and enforcing encapsulation through getters and setters. You learned how inheritance lets you reuse and extend behavior, and how polymorphism (using `virtual` and `override`) allows objects to take many forms. You also saw how access levels (private, public, protected) shape safe and predictable class design.

We covered how constructors initialize objects, how destructors handle cleanup, and how function and operator overloading make your classes more expressive. Through the banking system exercise, you applied everything by building an `Account` class with constructors, getters/setters, a display function, an enum for account types, and a menu loop that validated input and updated state safely.

You now have the tools to model real-world problems using clean, reusable, and maintainable OOP designs.

LESSON 9
Inheritance and Polymorphism

Topics Covered

☑ Base and derived class relationships and member inheritance.

☑ Understanding basic types of inheritance.

☑ Virtual functions enabling dynamic dispatch and runtime polymorphism.

☑ Using interfaces for polymorphic design.

Introduction

In this lesson, you will learn how inheritance creates an **is-a** relationship between base and derived classes, allowing shared data and behavior to be reused. You will explore different inheritance types, including public, protected, and private inheritance, as well as single, multilevel, multiple, hierarchical, and hybrid inheritance. You will also see how constructors chain across a class hierarchy and how access control works between base and derived classes to support clean class design and reduce duplication.

Next, we will introduce runtime polymorphism using virtual functions, the `override` keyword, dynamic dispatch, and the role of virtual destructors. Building on this, you will learn how abstract classes and interfaces are created using pure virtual functions.

Finally, you will apply these concepts in a Banking Menu app by creating `Checking` and `Savings` subclasses of `Account`, overriding `deposit`, `withdraw`, and `display`, enforcing rules through pure virtual functions, handling errors safely, and using smart pointers to call functions polymorphically.

EVALUATION COPY: Not to be used in class.

EVALUATION COPY: Not to be used in class.

9.1. Base and Derived Classes

Inheritance is a core concept of Object-Oriented Programming (OOP). It allows one class (called the derived class) to reuse the data and behavior of another class (called the base class).

This creates a clear **"is-a"** relationship and is implement using public inheritance in C++.

For example, an `Employee` **is a** `Person`.

Inheritance has two primary components:

- A **base class** is a class that serves as a starting point in object-oriented programming. Other classes can be built from it by inheriting its data members and member functions. Any class that inherits from a base class automatically gets access to its features and can also add new ones. A base class is often referred to as a **parent class** or **super class**.
- A **derived** class is a class that is formed using an existing class. It receives all accessible members and member functions of the base class and can extend them by adding new behavior or properties. A derived class is commonly called a **child class** or **subclass**.

C++ supports several inheritance models, but in this course we focus on the **is-a** inheritance model, which is by far the most common.

❖ 9.1.1. Syntax

To inherit from a class, use the `:` symbol:

```
class BaseClass{
   // additional members
}
class DerivedClass : public BaseClass{
   // additional members
}
```

Note that the semi-colon (`:`) symbol in class `DerivedClass : public BaseClass` is interpreted by the compiler as "inherits from".

❖ 9.1.2. Defining the Inheritance Hierarchy

To implement inheritance in C++, you define the derived class followed by a colon (`:`) and an access specifier (usually `public`) pointing to the base class.

We will use `Person` class as the base class and define two specialized derived classes, `Employee` and `Contractor`, which represent different types of people in a workforce. Both derived classes automatically gain access to the `Person`'s members (like `firstName` and `lastName`).

A Scenario: Persons in a Company

Imagine we are building a system for a company that works with both:

- **Employees:** They have a first name, last name, and an employee ID
- **Contractors:** They have a first name, last name, and a contracting company name

The first and last name are common to employees and contractors. Instead of repeating these variables in each derived class we define them once in the base class.

The Base Class: `Person`


```
class Person {
private:
    std::string firstName;
    std::string lastName;

public:
    // Constructor using first name and last name
    Person(const std::string& fName, const std::string& lName)
        : firstName(fName), lastName(lName) {}

    // Constructor using only last name
    Person(const std::string& lName)
        : firstName("Unknown"), lastName(lName) {}

    // Setters and getters
    void setFirstName(const std::string& fName) { firstName = fName; }
    void setLastName(const std::string& lName) { lastName = lName; }

    std::string getFirstName() const { return firstName; }
    std::string getLastName() const { return lastName; }
};
```

The Derived Class: `Employee`

```
class Employee : public Person {
private:
    int empId;

public:
    // Constructor using first name, last name, and employee ID
    Employee(const std::string& fName, const std::string& lName, int id)
        : Person(fName, lName), empId(id) {
    }
    // Constructor using last name, and employee ID
    Employee(const std::string& lName, int id)
        : Person(lName), empId(id) {
    }

    void setEmpId(int id) { empId = id; }
    int getEmpId() const { return empId; }
};
```

Note the following:

- `class Employee : public Person` tells the compiler that `Employee` **is a** type of `Person`.
- `Employee` inherits all the **public functions** of `Person`.
- `Employee` has **one member variable** named `empId`.
- Each constructor calls the `Person` **constructor** to set the first name and last name.
    - The C++ runtime environment creates a Person **subobject**.
    - The subobject constructor creates the member functions and variables of `Person`.
- `Employee` provides a **getter** and a **setter** for `empId`.

This avoids repeating shared code and allows us to extend behavior where needed.

The Other Derived Class: `Contractor`

```
class Contractor : public Person {
private:
    std::string contractorCompany;

public:
    // Constructor
    Contractor(const std::string& fName, const std::string& lName, const std::string&
company)
        : Person(fName, lName), contractorCompany(company) {
    }

   void setContractorCompany(const std::string& company) { contractorCompany = company;
 }
    std::string getContractorCompany() const { return contractorCompany; }
};
```

Items of interest:

- `Contractor` **inherits** from Person and therefore can use all of the public functions of `Person`.
- **One constructors** are defined and each one calls the `Person` constructor to set first name and last name.
- `Contractor` provides a **getter** and a **setter** for `contractorCompany`.

❖ 9.1.3. Using Derived Classes

The example below shows the usage of derived classes:

```
// create an Employee
Employee joeSmith("Joe", "Smith", 112233);
std::print("Employee {} {} has employee ID {}.\n",
           joeSmith.getFirstName(), joeSmith.getLastName(), joeSmith.getEmpId());

// create a Contractor
Contractor jenniferJones("Jennifer", "Jones", "West Coast IT Consulting LLC");
std::print("Contractor {} {} works for {}.\n",
             jenniferJones.getFirstName(),
             jenniferJones.getLastName(),
             jenniferJones.getContractorCompany());
```

Note the following:

- `Employee joeSmith` creates an `Employee` object.
- `Employee("Joe", "Smith", 112233)` calls its constructor.
- Even though neither `Employee` or Contractor defines `getFirstName()` or `getLastName()`, each class can use the functions because they came from `Person`.

Here is the output:

```
Employee Joe Smith has employee ID 112233.
Contractor Jennifer Jones works for West Coast IT Consulting LLC.
```

Items of interest:

- Shared variables, or fields, such as first name and last name are stored in the base class.
- Member variables that distinguish a subclass (e.g., employee ID or contractor company) are stored in the appropriate subclass.
- Inheritance promotes **reuse of code** because common attributes such as last name are stored in the base class only.

❖ 9.1.4. Difference between Base Class and Derived Class

Feature	Base Class	Derived Class
Definition	A class that provides properties and functions to other classes.	A class that is created using an existing class.
Functionality	Describes general behavior that can be shared.	Extends the base class by adding or modifying behavior.
Inheritance	Does not receive features from derived classes.	Receives accessible features from the base class.
Access	Controls which members can be inherited using access specifiers.	Can use the public and protected members of the base class.
Instantiation	Defined and declared before the derived class.	Written after the base class, specifying the visibility mode.

You can find the `Person` class and its derived classes in the following demo file.

Demo 9.1: InheritancePolymorphism/Demos/InheritanceAndPolymorphism.cpp

```
1.  #include <iostream>
2.  #include <format>
3.  #include <vector>
4.  #include <memory>
5.
6.  class Person {
7.  private:
8.      std::string firstName;
9.      std::string lastName;
10.
11. public:
12.     // Constructor with first name and last name
13.     Person(const std::string& fName, const std::string& lName)
14.         : firstName(fName), lastName(lName) {
15.     }
16.     // Overloaded constructor with last name only
17.     Person(const std::string& lName)
18.         :lastName(lName) {
19.         setFirstName("");
20.     }
21.
22.     // Getters
23.     std::string getFirstName() const { return firstName; }
24.     std::string getLastName() const { return lastName; }
25.
26.     // Setters
27.     void setFirstName(const std::string& fName) { firstName = fName; }
28.     void setLastName(const std::string& lName) { lastName = lName; }
29.
30.     // Destructor
31.     ~Person() {}
32.
33.     // virtual function
34.     virtual void display() {
35.         std::cout << std::format("Person with name {} {}.\n", getFirstName(),
             getLastName());
36.     }
37.     // pure virtual function (must be implemented in sublclasses)
38.     virtual std::string howPaid() = 0;
39.
40.     // overload == operator
41.     bool operator==(Person& anotherPerson) {
```

Demo 9.1: InheritancePolymorphism/Demos/InheritanceAndPolymorphism.cpp

```
42.         return (firstName == anotherPerson.firstName && lastName == anotherPer ↵↵
              son.lastName);
43.     }
44. };
45.
46. // create Employee subclass
47. class Employee : public Person {
48. private:
49.     int empId;
50.
51. public:
52.     Employee(const std::string& fName, const std::string& lName, int id)
53.         : Person(fName, lName), empId(id) {
54.     }
55.     Employee(const std::string& lName, int id)
56.         : Person(lName), empId(id) {
57.     }
58.
59.     void setEmpId(int id) { empId = id; }
60.     int getEmpId() const { return empId; }
61.
62.     void display()  override {   // override Person class display()
63.         Person::display();  // call Person class display()
64.         std::cout << std::format("An employee with employee ID {}.\n", getEm ↵↵
              pId()); // add Employee ID
65.     }
66.
67.     std::string howPaid() override {  // must be overriden here in subclass
68.         return "by W-2";
69.     }
70. };
71.
72. // create Contractor subclass
73. class Contractor : public Person {
74. private:
75.     std::string contractorCompany;
76.
77. public:
78.     Contractor(const std::string& fName, const std::string& lName, const
              std::string& company)
79.         : Person(fName, lName), contractorCompany(company) {
80.     }
81.
```

Demo 9.1: InheritancePolymorphism/Demos/InheritanceAndPolymorphism.cpp

```
82.       void setContractorCompany(const std::string& company) { contractorCompany =
              company; }
83.       std::string getContractorCompany() const { return contractorCompany; }
84.
85.       void display()  override {  // override Person class display()
86.           Person::display(); // call Person class display()
87.           // add Contractor Company:
88.          std::cout << std::format("A contractor working for {}.\n", getContractor  ↵↵
               Company());
89.       }
90.
91.       std::string howPaid() override { // must be overriden here in subclass
92.           return "by 1099";
93.       }
94.   };
95.
96.   int main() {
97.       // create an Employee
98.       Employee joeSmith("Joe", "Smith", 112233);
99.       joeSmith.display();
100.
101.      // create a Contractor
102.      Contractor jenniferJones("Jennifer", "Jones", "West Coast IT Consulting
              LLC");
103.      jenniferJones.display();
104.
105.      // demonstrate polymorphic behaviour of display()
106.      // first, create a vector of unique_ptr and type to Person (base class):
107.      std::vector<std::unique_ptr<Person>> persons;
108.
109.      persons.push_back(std::make_unique<Employee>(joeSmith)); // add Employee
              object to vector
110.     persons.push_back(std::make_unique<Contractor>(jenniferJones)); // add Con  ↵↵
              tractor object to vector
111.
112.      // use auto to adapt to the object type in the for loop
113.      std::cout << "\nPolymorphsim demo: Iterate the vector and call display on
              each object\n";
114.      for (auto& person : persons) {
115.          person->display();      // the correct display function is called at
               runtime
116.          // uncomment if you want to dereference the unique_ptr so that you an
               use dot notation to call display()
117.          //(*person).display();
```

Demo 9.1: InheritancePolymorphism/Demos/InheritanceAndPolymorphism.cpp

```
118.         std::cout << std::format("Paid {}.\n", person->howPaid());
119.     }
120. }
```


9.2. Inheritance Types

Inheritance in C++ allows a class to inherit member variables and functions from another class, enabling code reuse. The choice of inheritance type determines how public, protected, and private members can be accessed.

When a class inherits from another, we can control how the base class members are accessible in the derived class using three access specifiers: `public`, `protected`, and `private`.

Public Inheritance (`class Derived : public Base`)

Public inheritance is used when the derived class is a **specialization** of the base class. This type of inheritance represents a **"is-a"** relationship. For example, employee is a person.

This is by far the most common type of inheritance and we will explore it in detail.

.Accessibility in public Inheritance

Accessibility	`private` **members**	`protected` **members**	`public` **members**
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes

Protected Inheritance (`class Derived : protected Base`)

Protected inheritance is used to pass on the functionality of the base class to the the derived classes.

With protected inheritance, **public** base class members become **protected** in the derived class and are only accessible to derived classes. Protected inheritance **hides** a base class's public interface from users of the derived class.

Private Inheritance (`class Derived : private Base`)

Private inheritance hides the base class interface completely and expresses an "implemented-in-terms-of" relationship rather than "is-a."

All inherited members become **private**, and the base class is invisible to users of the derived class.

❖ 9.2.1. Inheritance by Hierarchy Type

Inheritance is not limited to just two classes (a base and a derived). Classes can be organized into complex structures, known as **inheritance hierarchies**. The way classes are arranged defines the type of inheritance being used, allowing us to model real-world relationships effectively.

Here are some of the types of inheritance that C++ supports:

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritance

In this course, we will study **single inheritance** in the exercises.

All forms of hierarchy type listed above are summarized below.

Single Inheritance

In single inheritance, a derived class is created using only **one parent class**. All classes derived from the base class have **only one** base class.

Single inheritance is the type of inheritance we have been using in the Person application. `Employee` and `Contractor` have one and only one base class which is `Person`.

Multilevel Inheritance

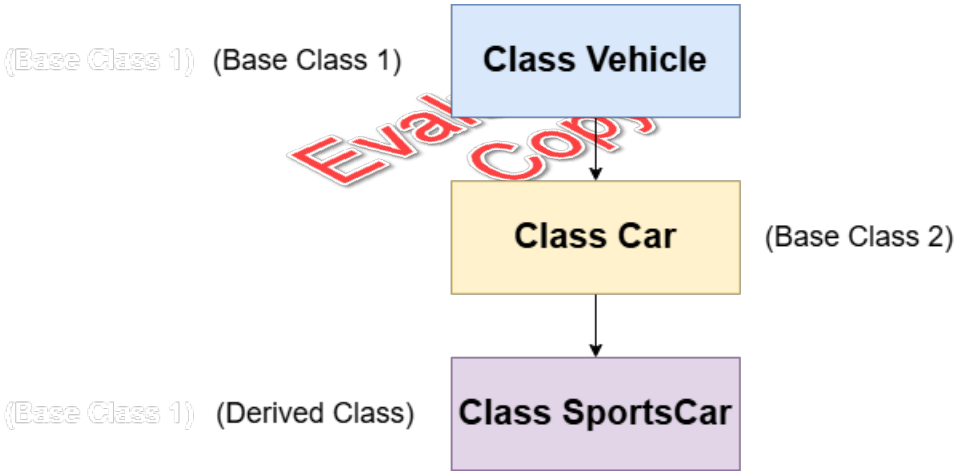
In multilevel inheritance, a class is derived from **another derived class**, forming a chain of inheritance. The class at the **top of the inheritance hierarchy** acts as the base class, and each subsequent class builds upon the one before it, allowing multiple levels of inheritance.

Multilevel Inheritance involves a **chain** of inheritance. A derived class inherits from a base class, and that derived class then acts as a base class for a new derived class.

For example a sports car is a specialized car, perhaps owning features like a high-performance engine that isn't available to for a regular car.

To paraphrase, we could say sports car is a car and car is a vehicle..

Figure: Multilevel inheritance, a class inherits from a derived class forming a chain.



Here are the classes definitions:

```
// Base Class
class Vehicle {
public:
    Vehicle() {
        std::cout << "Vehicle: standard features initialized\n";
    }
};

// Car inherits from Vehicle
class Car : public Vehicle {
public:
    Car() {
        std::cout << "Car: ready for road use\n";
    }
};

// Multilevel Inheritance
// SportsCar inherits from Car (Car already inherits Vehicle)
class SportsCar : public Car {
public:
    SportsCar() {
        std::cout << "SportsCar: high performance enabled\n";
    }
};
```


The following code creates a SportsCar object:

```
    std::cout << "\n--- Create a SportsCar ---\n";
    // Vehicle -> Car -> SportsCar
    SportsCar sc;
```

And here is the output:

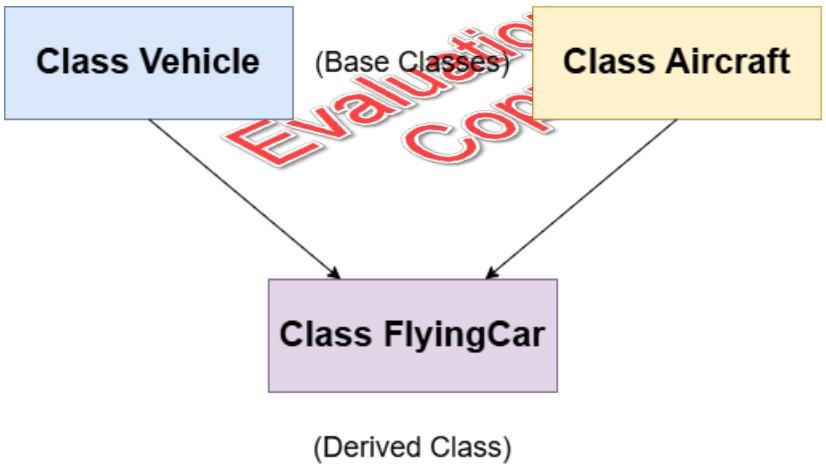
```
--- Create a SportsCar ---
Vehicle: standard features initialized
Car: ready for road use
SportsCar: high performance enabled
```

Multiple Inheritance

Multiple inheritance in C++ allows a single derived class to obtain variables and functions from **more than one** base class.

For example, a flying car can fly like an aircraft **and** can be driven like a car. Therefore, you could model a flying car be creating a `FlyingCar` subclass that inherits from **two base classes**, `Vehicle` and `Aircraft`.

Figure: Multiple inheritance, a class inherits from two or more base classes.



The following code implements our model:

```
class Vehicle {
public:
    Vehicle() {
        std::cout << "Vehicle: standard features initialized\n";
    }
};

class Aircraft {
public:
    Aircraft() {
        std::cout << "Aircraft ready for flight\n";
    }
};

// Derived class inheriting from both Vehicle AND Aircraft
class FlyingCar : public Vehicle, public Aircraft {
public:
    FlyingCar() {
        std::cout << "FlyingCar can drive and fly\n";
    }
};

int main() {
    // Creating an object of the derived class
    // calls constructors of all base classes
    FlyingCar fc;
    return 0;
}
```


Here is the output:

```
Vehicle: standard features initialized
Aircraft for flight
FlyingCar can drive and fly
```

You can try out the examples using the following demo file:

Demo 9.2: InheritancePolymorphism/Demos/InheritanceTypes.cpp

```
1.  #include <iostream>
2.
3.  // Base Class
4.  class Vehicle {
5.  public:
6.      Vehicle() {
7.          std::cout << "Vehicle: standard features initialized\n";
8.      }
9.  };
10.
11. // Car inherits from Vehicle
12. class Car : public Vehicle {
13. public:
14.     Car() {
15.         std::cout << "Car: ready for road use\n";
16.     }
17. };
18.
19. // Multilevel Inheritance
20. // SportsCar inherits from Car (Car already inherits Vehicle)
21. class SportsCar : public Car {
22. public:
23.     SportsCar() {
24.         std::cout << "SportsCar: high performance enabled\n";
25.     }
26. };
27.
28. class Aircraft {
29. public:
30.     Aircraft() {
31.         std::cout << "Aircraft for flight\n";
32.     }
33. };
34.
35. // Multiple Inheritance
36. // FlyingCar Is a Vehicle AND Is an Aircraft
37. class FlyingCar : public Vehicle, public Aircraft {
38. public:
39.     FlyingCar() {
40.         std::cout << "FlyingCar can drive and fly\n";
41.     }
42. };
43.
44.
```

Demo 9.2: InheritancePolymorphism/Demos/InheritanceTypes.cpp

```
45.  // Main Function
46.  int main() {
47.
48.      // Vehicle -> Car
49.      std::cout << "---  Create a Car ---\n";
50.      Car c;
51.
52.      std::cout << "\n--- Create a SportsCar ---\n";
53.      // Vehicle -> Car -> SportsCar
54.      SportsCar sc;
55.
56.      std::cout << "\n--- Create a FlyingCar ---\n";
57.      // Vehicle and Aircraft -> FlyingCar
58.      FlyingCar fc;
59.
60.      return 0;
61.  }
```


9.3. Virtual Functions and Dynamic Dispatch

Polymorphism in C++ is achieved through virtual functions and dynamic dispatch. This mechanism allows a single function call to execute different code depending on the actual type of the object being referenced at runtime.

In this lesson, you will learn how C++ decides which function to call at runtime using virtual functions. We will explain why normal function calls are resolved at compile time and how virtual functions enable **runtime polymorphism** when working with base class pointers and references.

❖ 9.3.1. What is a Virtual Function?

A **virtual function** is a member function of a base class that is marked with the keyword `virtual`. It tells the compiler that this function may be redefined in a derived class and should support runtime decision-making.

A virtual function allows a derived class to provide its own implementation of a function that already exists in the base class.

Why virtual functions are needed?

Without virtual functions:

- The function that gets called depends on the type of pointer or reference.

With virtual functions:

- The function that gets called depends on the actual object type.

This difference is what enables polymorphism. Virtual functions enable **runtime polymorphism**, calling the correct function via a base class pointer or reference.

Runtime polymorphism

Runtime polymorphism means:

The program decides which function to call while the program is running, not at compile time. The correct function is chosen at runtime, based on the object's actual type, not the type of the pointer or reference.

Key takeaways:

- You write code using a base class pointer (e.g., `Employee*`).
- The program decides at **runtime** which version of a virtual function to call.
- If the object is an `Employee`, the `Employee` implementation runs.
- If the object is a `Contractor`, the `Contractor` implementation runs.

This behavior is possible only when virtual functions are used.

We return to our to our `Person` as a base class and make the following observations:

- Every person has a first and last name.
- Some persons are employees that have an employee ID.
- Some persons are contractors that have a contractor company.

Now we posit the existence of a `display()` function for each class. We want **one** function call in the `main` function of our client program that **behaves differently** depending on the object.

This is exactly what virtual functions allow.

Syntax (Base Class)

The `virtual` keyword is required in the function declaration in base class definition.

```
class Person {
    // ... members ...
public:
    // virtual function
    virtual void display() {
       std::cout << std::format("Person with name {} {}.\n", getFirstName(), getLast ↵↵
Name());
    }
};
```

Placing `virtual` in the function header tells the compiler that this function can be overridden in derived classes.

Syntax (Derived Class)

The derived class provides its own implementation.

```
void display()  override {   // override Person class display()
     Person::display();  // call Person class display()
     std::cout << std::format("An employee with employee ID {}.\n", getEmpId()); // add
 Employee ID
}
```

Items of interest:

- The `override` keyword is a modern C++ feature used in the derived class function declaration.
- `override` tells the compiler that this function is meant to replace a virtual function from the base class.
- The base class's version of `display` is called using the scope resolution operator (`::`) to reuse the inherited logic.

`override` supplies an important safety mechanism. If you accidentally misspell the name, change the parameter list, or forget `const`, the compiler will generate an error because the function signature no longer matches the base class virtual function. This prevents accidentally **overloading** the function instead of overriding it.

❖ 9.3.2. Early Binding and Late Binding

When your program calls a function, C++ must decide which version of that function to run. This decision-making process is called **binding**. Depending on when this decision is made, binding can happen in two different ways:

Early Binding

Early binding occurs when the compiler decides which function to call at compile time. Since the decision is made before the program runs, function calls are faster and simpler. This is the default behavior for non-virtual functions.

Late Binding

Late binding happens at runtime and is used with virtual functions. The program waits until it is running to decide which function to execute, based on the actual type of the object, not the pointer or reference type. This extra decision step makes late binding slightly slower, but it enables runtime polymorphism.

❖ 9.3.3. Dynamic Dispatch

Dynamic Dispatch (also called **Late Binding** or **Runtime Binding**) is the process of deciding which function implementation to call while the program is executing. The function that gets executed is chosen at runtime, based on the actual object type, not the pointer type.

This happens only when using virtual functions.

- Dynamic dispatch only occurs when a virtual function is called through a pointer or reference to the **base class**.
- The C++ runtime environment ignores the type of the pointer (`Person*`) and instead looks at the **actual type of the object** it points to (`Employee` or `Contractor`) to select the correct function version.

When you call a virtual function through a **base class pointer**, C++ waits until the program is **running** to decide which version of the function to call, the base class one or the derived class one.

Why Do We Need Dynamic Dispatch?

Without dynamic dispatch:

- The base class function would always be called.
- Derived class behavior would be ignored.

With dynamic dispatch:

- Each object behaves correctly.
- One interface works for many object types.

Without dynamic dispatch, if you use a base class pointer (`Person*`) to point to a derived class object (`Employee`), a function call would default to the base class version (`Person::display()`), regardless of what the object actually is. Dynamic dispatch solves this, allowing for true polymorphism.

Example

```
Person* p = new Employee();
p->display();
```


- `p` looks like a `Person` pointer.
- The actual object is `Employee`.
- Dynamic dispatch ensures `Employee::display()` is called.

Without virtual functions, the base-class version would always be called.

How Dynamic Dispatch Works Internally

When a class has virtual functions, the compiler secretly adds two things to the class:

- **V-Pointer** (`vptr`)is a hidden pointer inside every object of that class (and its derived classes). The `vptr` points to the V-Table.
- **V-Table (Virtual Table)** is a static, secret lookup table (one per class) that contains the addresses of all the virtual functions for that specific class.
- Each object stores a hidden pointer to this table.
- The table contains addresses of overridden functions.
- At runtime, C++ uses the table to decide which function to call

You don't need to code this manually. C++ handles it automatically.

Dynamic Dispatch In Action

In the following example an `Employee` object and a `Contractor` object are created and then added to a vector. Then each vector item is processed in a for loop in which the polymorphic `display` and `howPaid` functions are called:

```
    Employee joeSmith("Joe", "Smith", 112233);
    Contractor jenniferJones("Jennifer", "Jones", "West Coast IT Consulting LLC");
    // demonstrate polymorphic behavior of display()
    // first, create a vector of unique_ptr and type to Person (base class):
    std::vector<std::unique_ptr<Person>> persons;
    persons.push_back(std::make_unique<Employee>(joeSmith)); // add Employee object to
vector
    persons.push_back(std::make_unique<Contractor>(jenniferJones)); // add Contractor
object to   vector

    // use auto to adapt to the object type in the for loop
   std::cout << "\nPolymorphsim demo: Iterate the vector and call display on each ob ↵↵
ject\n";
    for (auto& person : persons) {
        person->display();      // the correct display function is called at runtime
        // uncomment if you want to dereference the unique_ptr so that you an use dot
notation to call display()
        //(*person).display();
        std::cout << std::format("Paid {}.\n", person->howPaid());
    }
```


Items of interest:

1. The vector is defined as type `<std::unique_ptr<Person>>`. Using a smart pointer like `unique_ptr` enables polymorphism when calling functions in a derived class.
2. Each vector item is added as a `make_unique` pointer. We **make** a unique pointer and type it to either `Employee` or `Contractor` using a template.
3. The **appropriate constructor** is called when making the pointer.
4. Two **polymorphic functions** are called in the for loop:
    A. `person->display()`
    B. `person->howPaid()`

Here is the output. Note how the correct implementation of `display` and `howPaid` were called at runtime based on the **object type**:

```
Polymorphsim demo: Iterate the vector and call display on each object
Person with name Joe Smith.
An employee with employee ID 112233.
Paid by W-2.
Person with name Jennifer Jones.
A contractor working for West Coast IT Consulting LLC.
Paid by 1099.
```


❖ 9.3.4. Rules for Virtual Functions

- Virtual functions must be declared in the base class and they are overridden in derived class.
- Dynamic dispatch only works when a virtual function is called through a pointer or reference to the base class.
- The signature (name and parameter list) of the overriding function in the derived class must exactly match the base class's virtual function. Using `override` enforces this rule.
- Constructors and static member functions cannot be virtual.
- Function calls must be made using a base class pointer or reference.

You can test the examples in this activity using the following demo:

Demo 9.3: InheritancePolymorphism/Demos/InheritanceAndPolymorphism.cpp

```
1.  #include <iostream>
2.  #include <format>
3.  #include <vector>
4.  #include <memory>
5.
6.  class Person {
7.  private:
8.      std::string firstName;
9.      std::string lastName;
10.
11. public:
12.     // Constructor with first name and last name
13.     Person(const std::string& fName, const std::string& lName)
14.         : firstName(fName), lastName(lName) {
15.     }
16.     // Overloaded constructor with last name only
17.     Person(const std::string& lName)
18.         :lastName(lName) {
19.         setFirstName("");
20.     }
21.
22.     // Getters
23.     std::string getFirstName() const { return firstName; }
24.     std::string getLastName() const { return lastName; }
25.
26.     // Setters
27.     void setFirstName(const std::string& fName) { firstName = fName; }
28.     void setLastName(const std::string& lName) { lastName = lName; }
29.
30.     // Destructor
31.     ~Person() {}
32.
33.     // virtual function
34.     virtual void display() {
35.         std::cout << std::format("Person with name {} {}.\n", getFirstName(),
             getLastName());
36.     }
37.     // pure virtual function (must be implemented in sublclasses)
38.     virtual std::string howPaid() = 0;
39.
40.     // overload == operator
41.     bool operator==(Person& anotherPerson) {
```

Demo 9.3: InheritancePolymorphism/Demos/InheritanceAndPolymorphism.cpp

```
42.         return (firstName == anotherPerson.firstName && lastName == anotherPer ↵↵
              son.lastName);
43.     }
44. };
45.
46. // create Employee subclass
47. class Employee : public Person {
48. private:
49.     int empId;
50.
51. public:
52.     Employee(const std::string& fName, const std::string& lName, int id)
53.         : Person(fName, lName), empId(id) {
54.     }
55.     Employee(const std::string& lName, int id)
56.         : Person(lName), empId(id) {
57.     }
58.
59.     void setEmpId(int id) { empId = id; }
60.     int getEmpId() const { return empId; }
61.
62.     void display()  override {   // override Person class display()
63.         Person::display();  // call Person class display()
64.         std::cout << std::format("An employee with employee ID {}.\n", getEm ↵↵
              pId()); // add Employee ID
65.     }
66.
67.     std::string howPaid() override {  // must be overriden here in subclass
68.         return "by W-2";
69.     }
70. };
71.
72. // create Contractor subclass
73. class Contractor : public Person {
74. private:
75.     std::string contractorCompany;
76.
77. public:
78.     Contractor(const std::string& fName, const std::string& lName, const
              std::string& company)
79.         : Person(fName, lName), contractorCompany(company) {
80.     }
81.
```

Demo 9.3: InheritancePolymorphism/Demos/InheritanceAndPolymorphism.cpp

```
82.       void setContractorCompany(const std::string& company) { contractorCompany =
              company; }
83.       std::string getContractorCompany() const { return contractorCompany; }
84.
85.       void display()  override {  // override Person class display()
86.           Person::display(); // call Person class display()
87.           // add Contractor Company:
88.          std::cout << std::format("A contractor working for {}.\n", getContractor ↵↵
               Company());
89.       }
90.
91.       std::string howPaid() override { // must be overriden here in subclass
92.           return "by 1099";
93.       }
94.   };
95.
96.   int main() {
97.       // create an Employee
98.       Employee joeSmith("Joe", "Smith", 112233);
99.       joeSmith.display();
100.
101.      // create a Contractor
102.      Contractor jenniferJones("Jennifer", "Jones", "West Coast IT Consulting
              LLC");
103.      jenniferJones.display();
104.
105.      // demonstrate polymorphic behaviour of display()
106.      // first, create a vector of unique_ptr and type to Person (base class):
107.      std::vector<std::unique_ptr<Person>> persons;
108.
109.      persons.push_back(std::make_unique<Employee>(joeSmith)); // add Employee
              object to vector
110.     persons.push_back(std::make_unique<Contractor>(jenniferJones)); // add Con ↵↵
              tractor object to vector
111.
112.      // use auto to adapt to the object type in the for loop
113.      std::cout << "\nPolymorphsim demo: Iterate the vector and call display on
              each object\n";
114.      for (auto& person : persons) {
115.          person->display();      // the correct display function is called at
               runtime
116.          // uncomment if you want to dereference the unique_ptr so that you an
               use dot notation to call display()
117.          //(*person).display();
```

Demo 9.3: InheritancePolymorphism/Demos/InheritanceAndPolymorphism.cpp

```
118.         std::cout << std::format("Paid {}.\n", person->howPaid());
119.     }
120. }
```


9.4. Abstract Classes and Interfaces

While virtual functions enable dynamic behavior, Abstract Classes and Pure Virtual Functions take this concept further by enforcing a programming contract. They ensure that any derived class must provide its own specialized implementation for certain functions.

❖ 9.4.1. What Is an Abstract Class?

An **abstract class** is a class that cannot be **instantiated** (you cannot create objects of that class) on its own. It is designed to be used as a base class and must be inherited by other classes. A class becomes abstract when it contains at least one pure **virtual function**.

A pure virtual function defines what a derived class must do, but not how it will do that. This makes abstract classes ideal for defining common behavior that must be customized by derived classes.

Pure Virtual Function

A **pure virtual function** (also called a **pure virtual method**) is a virtual function that has no implementation in the base class.

It is declared by assigning `= 0` to the function. Therefore:

- The base class **cannot** provide a default implementation.
- Every derived class **must implement** this function.
- The base class becomes an **abstract class** and cannot be instantiated.

Pure virtual functions are used when the base class defines **what** must be done but not **how** it is done.

Syntax to Define an Abstract Class in C++

You make a class abstract by declaring a pure virtual function. This is done by adding `= 0` to the end of a virtual function declaration. This tells the compiler the function has no body in the base class.

A class becomes abstract when it contains a pure virtual function:

```
class abstractClassName {
public:
    // Pure virtual function
    virtual void functionName() = 0;
};
```

Items of interest:

- The `virtual` keyword marks the function as **virtual**.
- The function is equated to **0**. Now the function is a **pure** virtual function.
- Derived classes must implement the pure virtual function to avoid a compile error.
- Makes the entire class abstract

As an example, we will make the `Person` class abstract by introducing a pure virtual function named `howPaid()`:

```
virtual std::string howPaid() = 0;
```

By adding the pure virtual function `howPaid()` to the `Person` class, we make the class abstract. Persons must be paid somehow, but the base class doesn't know how.

This is a core part of **runtime polymorphism**. Derived classes will provide their own version of this function.

Why Do We Use Abstract Classes?

Abstract classes let us define a **common interface** for all subclasses.

For example:

- All persons (employees, contractors, interns, etc.) must have a `howPaid()` method.
- Each type will be paid differently.

- The parent class defines the requirement, and subclasses decide how to implement it.

❖ 9.4.2. Implementing a Pure Virtual Function in Subclasses

The following code shows the `howPaid` implementation in `Employee`:

```
std::string howPaid() override {
    return "by W-2";
}
```

and here is the `Contractor` implementation:

```
std::string howPaid() override {
    return "by 1099";
}
```

Each subclass supplies its own behavior for the same function. This is an example of **runtime polymorphism**.

Because `Person` is abstract, this line will cause a compile error:


```
// The following line will NOT compile because Person is abstract
Person person("Joe", "Smith");
```

You can only create objects of the derived classes (`Employee` or `Contractor`) because they are the concrete, or non-abstract, classes that provide the implementations of pure virtual functions.

When iterating through the vector of base class pointers, **the compiler guarantees** that every object has its own unique `howPaid()` method to call.

Here is an example of iterating through a vector containing employees and contractors:

```
for (auto& person : persons) {
    person->display();      // the correct display function is called at runtime
    std::print("Paid {}.\n", person->howPaid());  // call the pure virtual function
}
```

Note the following:

- `person->display()` calls the correct overridden version.
- `person->howPaid()` calls the correct overridden version.
- All decisions happen at runtime and illustrate the use of **dynamic dispatch**.

Here is the output:

```
Person with name Joe Smith.
An employee with employee ID 112233.
Paid by W-2.
Person with name Jennifer Jones.
A contractor working for West Coast IT Consulting LLC.
Paid by 1099.
```

❖ 9.4.3. What Is an Interface?

An **Interface** is an abstract class where **every function is a pure virtual function**. It contains no data members and no function implementations.

Interface acts purely as a contract or a blueprint for behavior. It guarantees that any class that implements the interface will implement a certain set of functions, without dictating how those methods are implemented.

An **interface** defines what a class *must do*, but not *how it does it*.

In C++, an interface is created using an abstract class where all functions are pure virtual:

- The class contains only **function declarations**.
- **None** of the functions provide an implementation.
- The interface acts like a **contract** that derived classes must follow

If you inherit from this class, you must implement these functions.

Interfaces help when:

- Default behavior is not tenable. For example, in the `Person` class we cannot provide a generic implementation of how a person is paid because the payment details vary between an employee and a contractor.
- You need to force subclasses to provide implementations for all functions defined in the base class.

❖ 9.4.4. Abstract Classes vs. Interfaces

The concept of an Interface is closely related to an abstract class.

Abstract Class	**Interface**
May contain implemented and pure virtual functions.	Contains only pure virtual functions (`= 0`).
Can have variables (like `firstName`).	No implementation logic.
Used to share common functionality.	Used to define strict behavior contracts.

The examples in this activity can be found in the following demo file.

Demo 9.4: InheritancePolymorphism/Demos/InheritanceAndPolymorphism.cpp

```
1.  #include <iostream>
2.  #include <format>
3.  #include <vector>
4.  #include <memory>
5.
6.  class Person {
7.  private:
8.      std::string firstName;
9.      std::string lastName;
10.
11. public:
12.     // Constructor with first name and last name
13.     Person(const std::string& fName, const std::string& lName)
14.         : firstName(fName), lastName(lName) {
15.     }
16.     // Overloaded constructor with last name only
17.     Person(const std::string& lName)
18.         :lastName(lName) {
19.         setFirstName("");
20.     }
21.
22.     // Getters
23.     std::string getFirstName() const { return firstName; }
24.     std::string getLastName() const { return lastName; }
25.
26.     // Setters
27.     void setFirstName(const std::string& fName) { firstName = fName; }
28.     void setLastName(const std::string& lName) { lastName = lName; }
29.
30.     // Destructor
31.     ~Person() {}
32.
33.     // virtual function
34.     virtual void display() {
35.         std::cout << std::format("Person with name {} {}.\n", getFirstName(),
             getLastName());
36.     }
37.     // pure virtual function (must be implemented in sublclasses)
38.     virtual std::string howPaid() = 0;
39.
40.     // overload == operator
41.     bool operator==(Person& anotherPerson) {
```

Demo 9.4: InheritancePolymorphism/Demos/InheritanceAndPolymorphism.cpp

```
42.         return (firstName == anotherPerson.firstName && lastName == anotherPer ↵↵
              son.lastName);
43.     }
44. };
45.
46. // create Employee subclass
47. class Employee : public Person {
48. private:
49.     int empId;
50.
51. public:
52.     Employee(const std::string& fName, const std::string& lName, int id)
53.         : Person(fName, lName), empId(id) {
54.     }
55.     Employee(const std::string& lName, int id)
56.         : Person(lName), empId(id) {
57.     }
58.
59.     void setEmpId(int id) { empId = id; }
60.     int getEmpId() const { return empId; }
61.
62.     void display()  override {   // override Person class display()
63.         Person::display();  // call Person class display()
64.         std::cout << std::format("An employee with employee ID {}.\n", getEm ↵↵
              pId()); // add Employee ID
65.     }
66.
67.     std::string howPaid() override {  // must be overriden here in subclass
68.         return "by W-2";
69.     }
70. };
71.
72. // create Contractor subclass
73. class Contractor : public Person {
74. private:
75.     std::string contractorCompany;
76.
77. public:
78.     Contractor(const std::string& fName, const std::string& lName, const
              std::string& company)
79.         : Person(fName, lName), contractorCompany(company) {
80.     }
81.
```

Demo 9.4: InheritancePolymorphism/Demos/InheritanceAndPolymorphism.cpp

```
82.       void setContractorCompany(const std::string& company) { contractorCompany =
              company; }
83.       std::string getContractorCompany() const { return contractorCompany; }
84.
85.       void display()  override {  // override Person class display()
86.           Person::display(); // call Person class display()
87.           // add Contractor Company:
88.          std::cout << std::format("A contractor working for {}.\n", getContractor ↵↵
               Company());
89.       }
90.
91.       std::string howPaid() override { // must be overriden here in subclass
92.           return "by 1099";
93.       }
94.   };
95.
96.   int main() {
97.       // create an Employee
98.       Employee joeSmith("Joe", "Smith", 112233);
99.       joeSmith.display();
100.
101.      // create a Contractor
102.      Contractor jenniferJones("Jennifer", "Jones", "West Coast IT Consulting
              LLC");
103.      jenniferJones.display();
104.
105.      // demonstrate polymorphic behaviour of display()
106.      // first, create a vector of unique_ptr and type to Person (base class):
107.      std::vector<std::unique_ptr<Person>> persons;
108.
109.      persons.push_back(std::make_unique<Employee>(joeSmith)); // add Employee
              object to vector
110.     persons.push_back(std::make_unique<Contractor>(jenniferJones)); // add Con ↵↵
              tractor object to vector
111.
112.      // use auto to adapt to the object type in the for loop
113.      std::cout << "\nPolymorphsim demo: Iterate the vector and call display on
              each object\n";
114.      for (auto& person : persons) {
115.          person->display();      // the correct display function is called at
               runtime
116.          // uncomment if you want to dereference the unique_ptr so that you an
               use dot notation to call display()
117.          //(*person).display();
```

Demo 9.4: InheritancePolymorphism/Demos/InheritanceAndPolymorphism.cpp

```
118.         std::cout << std::format("Paid {}.\n", person->howPaid());
119.     }
120. }
```

Exercise 9: Inheritance and Polymorphism Practice

30 to 45 minutes

In this exercise, you will create `Checking` and `Savings` subclasses of `Account` in the Banking Menu app.

❖ E9.1. Requirements

1. Create two subclasses of `Account`.
    A. `Checking` will have a boolean member variable that specifies whether the holder wants paper checks.
    B. `Savings` will have a `double` member variable that stores the transaction fee for each savings withdrawal.
2. The member variables distinguish `Checking` and `Savings` from each other.
3. Both subclasses will share the account number, holder name, and balance defined in `Account` class.
4. Prompt the user in a while loop to choose which type of account to create.

❖ E9.2. Step-by-step Instructions

1. Open `InheritanceAndPolymorphism\Exercises\BankingMenu.cpp` in the editor. This is your starter file.
2. Remove the `AccountType` enum from the `Account` class.
3. Remove or comment out the `deposit` and `withdraw` functions from `Account`. These operations will now be handled by member functions in the `Account` subclasses.
4. Add a virtual function named `display` to the `Account` class:
    A. The function should return a `std::string`.
    B. The string that is returned should contain the account number, holder name, and balance in a meaningful sentence.
5. Add two pure virtual functions to the `Account` class:

A. `deposit(double amount)`

B. `withdraw(double amount)`

C. Both functions should return `void`.

6. Create a subclass of `Account` named `Checking`:

   A. Add one `bool` member variable named `paperChecks`. If the value is `true`, then the account holder wants a checkbook with paper checks. Otherwise, the holder does not want a checkbook.

   B. Write a constructor that accepts:

      i. Account number

      ii. Holder name

      iii. Paper checks flag

      iv. Remember to call the `Account` constructor with account number and holder.

   C. Write a member function named `deposit` that overrides the pure virtual function in `Account`:

      i. Increase the balance by adding the deposit amount passed to the function.

   D. Write another member function named `withdraw`:

      i. Decrease the balance by subtracting the withdrawal amount passed to the function.

   E. Write a `display` function that:

      i. Calls the `Account` class `display` function.

      ii. Append information about whether paper checks are enabled. Then print the value of `paperChecks`.

7. Create a subclass of Account named `Savings`.

   A. Add the following private member variables:

      i. `double transactionFee`

      ii. `int nbrOfWithdrawals`, initialized to `0`

   B. Define the following public constants:

      i. `SAVINGS_TRANSACTION_FEE` with a value of `2.50`. The fee should be subtracted from the balance for each withdrawal.

ii. `MAX_SAVINGS_WITHDRAWALS` with a value of `3`. When the number of withdrawals exceeds this number, then a fee will be charged (see the next constant below).

iii. `SAVINGS_MAX_WD_FEE` with a value of `3.00`. This fee should be subtracted from the balance for each withdrawal after `MAX_SAVINGS_WITHDRAWALS` has been exceeded.

C. Write two constructors:

i. One that accepts account number, holder name, and transaction fee.

ii. One that accepts account number and holder name and uses the default transaction fee.

D. Override the `deposit` function:

i. Increase the balance by the deposit amount.

E. Override the `withdraw` function:

i. Throw a `std::runtime_error` if the withdrawal amount exceeds the balance.

ii. Increment `nbrOfWithdrawals` by 1.

iii. If `nbrOfWithdrawals` exceeds `MAX_SAVINGS_WITHDRAWALS`, increase the withdrawal amount by `SAVINGS_MAX_WD_FEE`.

iv. Subtract the withdrawal amount and transaction fee from the balance.

F. Override the `display` function:

i. Call the `Account` class `display` function.

ii. Append the transaction fee information.

8. In the main function, Prompt the user for the account holder name and store it in `accountHolder`.

9. Create an object of type `Account`. You will be using some functions polymorphically, therefore use a `unique_ptr` when creating the object. For example `std::unique_ptr<Account> account;`

10. Prompt the user in a while loop to choose the account type.

11. If the user selects a `Checking` account:

A. Ask whether paper checks are required.

B. Create the account using `std::make_unique<Checking>` with a hard-coded account number.

12. If the user selects a `Savings` account:

    A. Prompt the user for the transaction fee for each withdrawal. If the users enters `d` (character 'd') then use `SAVINGS_TRANSACTION_FEE` as the transaction fee when creating the `Savings` object (**Hint**: call the correct `Savings` constructor that assigns the default transaction fee) else use the value entered by the user. Use exception handling to ensure that the value is numeric and is not negative.

    B. Set the `account` object equal to a `make_unique` pointer. Use `<Savings>` to establish the pointer type.

    C. When calling one of the `Savings` constructors:

        i. Hard code the account number, e.g., `555666777`. Use `accountHolder` as the holder.

        ii. If the user entered a transaction fee then pass the user-supplied value to the constructor. You will be calling the constructor that expects account number, holder, and transaction fee.

        iii. If the user did not enter a transaction fee then call the `Savings` constructor that expects account number and holder only.

13. In the menu "do-while" loop:

    A. Call `deposit` or `withdraw` on the account object based on user input. For example,

        i. If the user wants to deposit money into a checking account, then call the `deposit` function for the `Checking` object.

        ii. If the user wants to withdraw money from a savings account, then call the `withdrawal` function for the `Savings` object.

14. After the user exits the loop:

    A. Print the transaction logs.

    B. Call the `display` function to print the `account` object's variables.

Solution: InheritancePolymorphism/Solutions/BankingMenu.cpp

```
1.  #include <iostream>
2.  #include <memory>
3.  #include <format>
4.  #include <stdexcept>
5.  #include <string>
6.  #include <vector>
7.  #include <map>
8.  #include <deque>
9.  #include <algorithm>
10.
11. typedef std::string str;
12.
13. const double START_BALANCE = 0.0; // Initial account balance
14.
15. str userInput;
16.
17. class Account {
18.
19. private:
20.     int acctNbr;
21.     str holder;
22.     double balance;
23.
24. protected:
25.     void setBalance(double balance) { this->balance = balance; }
26.
27. public:
28.     Account(int acctNbr, str holder, double balance)
29.         : acctNbr(acctNbr), holder(holder), balance(balance) {
30.     }
31.     // default constructor (no parameters)
32.     Account() {
33.         acctNbr = 999999999;
34.         holder = "N/A";
35.         balance = 0.0;
36.     }
37.     // Getters
38.     int getAcctNbr() { return acctNbr; }
39.     str getHolder() { return holder; }
40.     double getBalance() { return balance; }
41.
42.     // Setters
43.     void setAcctNbr(int acctNbr) { this->acctNbr = acctNbr; }
44.     void setHolder(str holder) { this->holder = holder; }
```

Solution: InheritancePolymorphism/Solutions/BankingMenu.cpp

```
45.
46.      // virtual display function
47.      virtual std::string display() {
48.          return std::format("Account number {} with holder {} and balance of
          ${:.2f}.\n",
49.             getAcctNbr(), getHolder(), getBalance());
50.      }
51.
52.      //pure virtual functions
53.      virtual void deposit(double amount) = 0;
54.
55.      virtual void withdraw(double amount) = 0;
56.
57.  };
58.
59.  class Checking : public Account {
60.  private:
61.      bool paperChecks;
62.  public:
63.      Checking(int acctNbr, str holder, double balance, bool paperChecks)
64.          : Account(acctNbr, holder, balance), paperChecks(paperChecks) {
65.      }
66.      void deposit(double amount) override {
67.          if (amount <= 0) {
68.              std::string error_msg =
69.                  std::format("Deposit rejected: amount ${:.2f} is less than or
           equal to zero.\n", amount);
70.              throw std::runtime_error(error_msg);
71.          }
72.          double balance = getBalance();
73.          setBalance(balance + amount);
74.      }
75.      void withdraw(double amount) override {
76.          if (amount > getBalance()) {
77.              throw std::runtime_error("Withdrawal amount exceeds balance");
78.          }
79.          else if (amount <= 0) {
80.              std::string error_msg =
81.                  std::format("Withdrawal rejected: amount ${:.2f} is less than
           or equal to zero.\n", amount);
82.              throw std::runtime_error(error_msg);
83.          }
84.          double balance = getBalance();
85.          setBalance(balance - amount);
86.      }
```

Solution: InheritancePolymorphism/Solutions/BankingMenu.cpp

```
87.      // virtual display function
88.      virtual std::string display() {
89.          return Account::display() + std::format("Paper checks for this Checking
               account is {}.\n", paperChecks);
90.      }
91.  };
92.
93.  class Savings : public Account {
94.  private:
95.      double transactionFee;
96.      // track number of withdrawals for each Savings object:
97.      int nbrOfWithdrawals = 0;
98.  public:
99.      // Transaction fee for a Savings withdrawal
100.     const double SAVINGS_TRANSACTION_FEE = 2.50;
101.     // Fee for exceeding maximum number of Savings withdrawals:
102.     const double SAVINGS_MAX_WD_FEE = 3.00;
103.     // Maximum number of savings withdrawals without fee:
104.     const int MAX_SAVINGS_WITHDRAWALS = 3;
105.
106.     // Constructor with transaction fee
107.     Savings(int acctNbr, str holder, double balance, double transactionFee)
108.         : Account(acctNbr, holder, balance), transactionFee(transactionFee) {
109.     }
110.     // Constructor without transaction fee...default transaction fee will be
           used
111.     Savings(int acctNbr, str holder, double balance)
112.         : Account(acctNbr, holder, balance) {
113.             transactionFee = SAVINGS_TRANSACTION_FEE;
114.     }
115.
116.     void deposit(double amount) override {
117.         if (amount <= 0) {
118.             std::string error_msg =
119.                 std::format("Deposit rejected: amount ${:.2f} is less than or
              equal to zero.\n", amount);
120.             throw std::runtime_error(error_msg);
121.         }
122.         double balance = getBalance();
123.         setBalance(balance + amount);
124.     }
125.     void withdraw(double amount) override {
126.         if (amount > getBalance()) {
127.             throw std::runtime_error("Withdrawal amount exceeds balance");
128.         }
```

Solution: InheritancePolymorphism/Solutions/BankingMenu.cpp

```
129.         else if (amount <= 0) {
130.             std::string error_msg =
131.                 std::format("Withdrawal rejected: amount ${:.2f} is less than
              or equal to zero.\n", amount);
132.             throw std::runtime_error(error_msg);
133.         }
134.         else {
135.             nbrOfWithdrawals++;
136.             if (nbrOfWithdrawals > MAX_SAVINGS_WITHDRAWALS) {
137.                 amount += SAVINGS_MAX_WD_FEE;
138.             }
139.     }
140.         double balance = getBalance() - transactionFee;
141.         setBalance(balance - amount);
142.     }
143.     // virtual display function
144.     virtual std::string display() {
145.        return Account::display() + std::format("Transaction fee for this Savings
              account is ${:.2f}.\n", transactionFee);
146.     }
147. };
148.
149. int main() {
150.
151.     std::vector<double> depositLog{}; // use vector for deposits
152.     std::deque<double> withdrawDeque; // use deque for withdrawals
153.     std::map<std::string, double> transactionsMap;
154.     // set total deposits to 0
155.     transactionsMap["Deposits"] = 0;
156.     // set total withdrawals to 0
157.     transactionsMap["Withdrawals"] = 0;
158.
159.     // prompt user for account holder name:
160.     std::cout << "Please enter account holder name: ";
161.     getline(std::cin, userInput);
162.     str accountHolder = userInput;
163.     // create the Account object:
164.     std::unique_ptr<Account> account;
165.
166.     bool correct = false;
167.     // prompt the user for account type:
168.     while (!correct && userInput[0] != 'Q') {
169.         bool continueLooping = true; // used in the 'S' case of the switch
              statement
```

Solution: InheritancePolymorphism/Solutions/BankingMenu.cpp

```
170.          std::cout << "Enter account type (C for Checking, S for Savings) or Q
               to Quit the app: ";
171.          getline(std::cin, userInput);
172.          switch (userInput[0]) {
173.          case 'C':
174.          case 'c':
175.              std::cout << "Do you want paper checks (Y/N)? ";
176.              getline(std::cin, userInput);
177.              if (userInput[0] == 'Y' || userInput[0] == 'y') {
178.                  account = std::make_unique<Checking>(555666777, accountHolder,
              START_BALANCE, true);
179.              }
180.              else {
181.                  account = std::make_unique<Checking>(555666777, accountHolder,
              START_BALANCE, false);
182.              }
183.              correct = true;
184.              break;
185.          case 'S':
186.          case 's':
187.              do {
188.                  std::cout << "Enter the transaction fee for each withdrawal or
              \'d\' for the default fee ($2.50): ";
189.                  try {
190.                      getline(std::cin, userInput);
191.                      if (userInput[0] == 'd' || userInput[0] == 'D') {
192.                          // user wants to use the default fee
193.                        account = std::make_unique<Savings>(555666777, accoun ↵↵
              tHolder, START_BALANCE);
194.                      }
195.                      else {
196.                          double userTransactionFee = std::stod(userInput);  //
              get user transaction fee
197.                          if (userTransactionFee < 0) {
198.                              // throw invalid argument exception with message
              place holder
199.                              // (hard coded message will be displayed in catch
              block)
200.                              throw std::invalid_argument("message");
201.                          }
202.                       account = std::make_unique<Savings>(555666777, accoun ↵↵
              tHolder, START_BALANCE, userTransactionFee);
203.                      }
204.                      continueLooping = false;
205.                  }
206.                  catch (std::invalid_argument& ie) {
```

Solution: InheritancePolymorphism/Solutions/BankingMenu.cpp

```
207.                     std::cout << "You entered a non-numeric or negative fee.
             Please try again.\n";
208.                 }
209.             }
210.             while (continueLooping);
211.             correct = true;
212.             break;
213.         case 'Q':
214.         case 'q':
215.             std::cout << "Closing the app by user request.\n";
216.             return 0;
217.         default:
218.            std::cout << "Unknown account type...please re-enter (Q to quit).\n";
219.         }
220.     }
221.
222.     do {
223.         // Display menu options
224.         std::cout << " Deposit (d)\n";
225.         std::cout << " Withdraw (w)\n";
226.         std::cout << " Balance (b)\n";
227.         std::cout << " Quit (q)\n\n";
228.         std::cout << "Enter choice: ";
229.         std::getline(std::cin, userInput);
230.         switch (userInput[0]) {
231.             // Handle deposit
232.             case 'd':
233.             case 'D': {
234.                 double amount;
235.                 std::cout << "Enter amount to deposit: ";
236.                 std::getline(std::cin, userInput);
237.                 try {
238.                     amount = std::stod(userInput);
239.                     account->deposit(amount);
240.                     // log deposit amount to vector
241.                     depositLog.push_back(amount);
242.                     // add deposit amount to running total on the map
243.                     transactionsMap["Deposits"] += amount;
244.                     std::cout << std::format("Deposited ${:.2f}\n", amount);
245.                 }
246.                 catch (std::invalid_argument& ie) {
247.                    std::cout <<  "Deposit amount must be numeric. Please try
             again.\n";
248.                 }
```

Solution: InheritancePolymorphism/Solutions/BankingMenu.cpp

```
249.                 catch (std::runtime_error& re) {
250.                    std::cout << re.what();
251.                 }
252.                 break;
253.             }
254.             // Handle withdrawal
255.             case 'w':
256.             case 'W': {
257.                 double amount;
258.                 std::cout << "Enter withdrawal amount: ";
259.                 std::getline(std::cin, userInput);
260.                 try {
261.                     amount = std::stod(userInput);
262.                     account->withdraw(amount);
263.                     // log withdrawal to deque
264.                     // use push_front so that most recent withdrawal is front
            of the deque
265.                     withdrawDeque.push_front(amount);
266.                     // add withdrawal amount to running total on the map
267.                     transactionsMap["Withdrawals"] += amount;
268.                     std::cout << std::format("Withdrew ${:.2f}\n", amount);
269.                 }
270.                 catch (std::invalid_argument& ie) {
271.                  std::cout << "Withdrawal amount must be numeric, amount re ↵↵
            jected.\n";
272.                 }
273.                 catch (std::runtime_error& rte) {
274.                     std::cout << rte.what();
275.                 }
276.                 break;
277.             }
278.             // Display balance
279.             case 'b':
280.             case 'B':
281.                 std::cout << std::format("Current balance: ${:.2f}\n", account-
            >getBalance());
282.                 break;
283.             // Handle quit
284.             case 'q':
285.             case 'Q':
286.                 std::cout << std::format("Final balance: ${:.2f}\n", account-
            >getBalance());
287.                 break;
288.             // Handle invalid input
289.             default:
```

Solution: InheritancePolymorphism/Solutions/BankingMenu.cpp

```
290.                 std::cout << "Invalid selection. Please choose a valid option.\n";
291.             }
292.         } while (userInput[0] != 'q' && userInput[0] != 'Q');
293.
294.         // print withdrawDeque as a stack so more recent withdrawals are displayed
                 first
295.         std::cout << std::format("{:^20}\n", "Log of Withdrawals-Most Recent First");
296.         for (const auto& wd : withdrawDeque) {
297.             std::cout << std::format("{:>20}\n", std::format("${:.2f}", wd));
298.         }
299.         // print withdrawDeque as a queue so least recent withdrawals are displayed
                 first
300.         std::cout << std::format("{:^20}\n", "Log of Withdrawals-Least Recent First");
301.         for (auto wdIter = withdrawDeque.rbegin(); wdIter != withdrawDeque.rend();
                 wdIter++) {
302.             std::cout << std::format("{:>20}\n", std::format("${:.2f}", *wdIter));
303.         }
304.         // print deposit log
305.         std::cout << std::format("{:^20}\n", "Log of Deposits");
306.         std::sort(depositLog.begin(), depositLog.end(), std::greater<>{});  // sort
                 deposits in descending sequence
307.         for (auto& deposit : depositLog) {
308.             if (deposit == 0) {
309.                 break;
310.             }
311.             std::cout << std::format("{:>20}\n", std::format("${:.2f}", deposit));
312.         }
313.         // print account information
314.         std::cout << std::format("{}\n", account->display());
315.         // print totals from transactions map
316.         std::cout << std::format("Total amount of deposits is ${:.2f}.\n", transac ↵↵
                 tionsMap["Deposits"]);
317.         std::cout << std::format("Total amount of withdrawals is ${:.2f}.\n",
                 transactionsMap["Withdrawals"]);
318. }
```


Conclusion

In this lesson, you learned how to model **is-a** relationships using base and derived classes, reuse shared data and behavior, and connect constructors across an inheritance hierarchy. You explored inheritance by access type (public, protected, and private) and by hierarchy level (single, multilevel, multiple, hierarchical, and hybrid), and saw how each choice affects visibility and class design.

You also practiced runtime polymorphism using virtual functions, the `override` keyword, and dynamic dispatch through base class pointers and references, and learned why virtual destructors are important.

Finally, you applied these concepts in the Banking Menu exercise by implementing an abstract `Account` class with `Checking` and `Savings` specializations, using `unique_ptr` for polymorphic behavior, and enforcing rules such as transaction fees and withdrawal limits. You are now prepared to design clean, extensible class hierarchies and apply polymorphism effectively in real programs.

LESSON 10
File I/O

Topics Covered

☑ What file streams are and how they work.

☑ Reading from files with ifstream.

☑ Writing and appending to files with ofstream.

☑ Choosing and combining file open modes.

☑ Checking stream state and handling I/O errors or exceptions.

☑ Parsing structured data files.

Introduction

In this lesson, you will learn how to use file I/O (input/output) in C++ so that you can read and write text files.

You will learn how to choose file modes such as `in`, `out`, `app`, `trunc` and how to check for errors using functions such as `good()`, `fail()`, `bad()`, and `eof()`.

A variety of demonstration files are provided to help you learn how to apply file I/O in your C++ programs.

EVALUATION COPY: Not to be used in class.

EVALUATION COPY: Not to be used in class.

10.1. File Streams and Concepts

In C++, a *stream* is a general way for programs to handle input and output. Up to now, you have worked with streams, such as `std::cin` for keyboard input and `std::cout` for screen output. These streams let data "flow" into and out of your program.

❖ 10.1.1. What are File Streams?

A **file stream** is simply a stream that reads data from a file or writes data to a file instead of using the keyboard or screen. The idea is the same: a file becomes just another source or destination for data.

Think of a file stream as a "bridge" between your C++ program and a file stored on the computer. Once this bridge is created, you can treat reading and writing almost the same way you treat `cin` and `cout`.

❖ 10.1.2. Types of File Streams

C++ provides three main file-related stream types:

- `fstream` - can both read from *and* write to a file.
- `ifstream` - reads from a file (input file stream).
- `ofstream` - writes to a file (output file stream).

Each type of file stream gives your program a way to communicate with a file.

❖ 10.1.3. Including the File Stream Library

To work with files, C++ provides a special header:

```
#include <fstream>
```

This header supplies the file-stream classes you will use later.

❖ 10.1.4. Why Use File Streams?

File streams are powerful because they allow programs to:

- Save results permanently.

- Load and process data from external files.
- Read configuration settings.
- Store logs or reports.
- Handle large datasets that don't come from user input.

Files are essential in real-world applications, and file streams make them easy to use in C++.

❖ 10.1.5. Opening and Closing Files

Before using a file, a stream must be connected to it. This is called **opening** the file.

After finishing reading or writing data, the file should be closed, which disconnects the stream and ensures buffered data is written to disk.

Later, you'll learn actual syntax like:

```
file.open("example.txt");
file.close();
```

The key idea for now is simply: **open use close.**

❖ 10.1.6. Using File Streams like Regular Streams

One of the biggest advantages of C++ file streams is that the same operators used with `cin` and `cout` also work with file streams:

- `<<` for sending data *to* a stream.
- `>>` for taking data *from* a stream.

This means once a file stream is opened, interacting with it feels very familiar.

❖ 10.1.7. File Stream Objects

A file stream object is just a variable that represents a connection to a file. For example:

```
std::fstream file;
```

This object will later be used to open a file, read from it, write to it, and close it.

❖ 10.1.8. Concept Summary

- A **stream** is a path for data to flow in (input) or out (output).
- A **file stream** lets programs work with files the same way they work with console input/output.
- C++ provides `fstream`, `ifstream`, and `ofstream` for file operations.
- You must open a file stream before using it and close it afterward.
- The same `<<` and `>>` operators used with `cout` and `cin` also work with file streams.

10.2. Reading Files: ifstream

When you want your C++ program to read data stored in a file, such as numbers, text, names, or settings, you can use the `ifstream` class. `ifstream` stands for input file stream, and it is used to read data *from* a file, just like `cin` reads data from the keyboard.

❖ 10.2.1. What is `ifstream`?

`ifstream` is a special kind of stream designed to read data from files. Once a file is opened through an `ifstream` object, you can use:

- `>>` to read words or numbers.
- `getline()` to read entire lines.

Just like `cin`, but from a file.

To use `ifstream`, you must include:

```
#include <fstream>   // Provides ifstream
```

❖ 10.2.2. Opening a File with `ifstream`

There are two common ways to open a file:

Method 1: Declare and Open Separately

```
std::ifstream file;       // Create file stream object
file.open("data.txt");    // Open file named "data.txt"
```

Method 2: Open during Declaration

```
std::ifstream file("data.txt");  // Opens the file immediately
```

Notes:

- Use a relative path (like "data.txt") or an absolute path (like "C:/files/data.txt").
- The file must exist for reading; ifstream will not create it.

❖ 10.2.3. Reading From a File

Below are examples demonstrating different reading techniques:

Example 1: Reading Words or Numbers Using >>

We can use >> operator to read words or numbers from a file.

```
#include <iostream>
#include <fstream>

int main() {
    // Create an ifstream object and try to open "data.txt"
    std::ifstream file("data.txt");

    // Check if the file failed to open
    if (!file) {
        std::cout << "Could not open file.\n";
        return 1;   // Exit the program if opening failed
    }

    int num;
    // Read the first number (or word) from the file into 'num'
    file >> num;

    // Output the value that was read
    std::cout << "Number read: " << num << std::endl;

    // Close the file after finishing
    file.close();

    return 0;
}
```


If **data.txt** contains 42, this will print:

Output

```
Number read: 42
```

Example 2: Read Full Lines Using `getline()`

`getline()` is used when you want to read whole sentences or lines.

```
#include <iostream>
#include <fstream>
#include <string>

int main() {
    // Create an ifstream object and open "message.txt"
    std::ifstream file("message.txt");

    // Check if the file failed to open
    if (!file) {
        std::cout << "File not found.\n";
        return 1;   // Exit the program if opening failed
    }

    std::string line;

    // Read the file line by line until reaching the end
    while (std::getline(file, line)) {
        std::cout << line << std::endl;   // Print each line to the screen
    }

    // Close the file after finishing
    file.close();

    return 0;
}
```


This is useful when reading sentences, paragraphs, or structured text files.

❖ 10.2.4. Reading Until End of File

File reading often uses a loop like:

```
while (file >> value) {   // Repeats as long as reading is successful
    // process value
}
```

This loop ends automatically when there is no more data to read.

❖ 10.2.5. Closing the File

Files should always be closed:

```
file.close();   // Frees resources and completes file operations
```

❖ 10.2.6. Key Points to Remember

- Use `ifstream` for reading files.
- Include `<fstream>`.
- Always check if the file opened successfully.
- Use `>>` for simple values and `getline()` for whole lines.
- Always close the file when finished.

Demo 10.1: FileIo/Demos/ifstream.cpp

```
1.   #include <iostream>
2.   #include <fstream>
3.
4.   int main() {
5.       // Create an ifstream object and open "numbers.txt"
6.       std::ifstream inputFile("numbers.txt");
7.
8.       // Check if the file was opened successfully
9.       if (!inputFile) {
10.          std::cout << "Error: Could not open file 'numbers.txt'.\n";
11.          return 1; // Exit the program if the file cannot be opened
12.      }
13.
14.      int number; // Variable to store each number read from the file
15.
16.      // Read numbers from the file one by one until the end of file
17.      while (inputFile >> number) {
18.          std::cout << "Number read: " << number << std::endl;
19.      }
20.
21.      // Close the file when done
22.      inputFile.close();
23.
24.      return 0;
25.  }
```

❖ 10.2.7.

10.3. Writing Files: ofstream

When you want your C++ program to write data to a file you will use the `ofstream` class. `ofstream` stands for output file stream, and it is used to *write* data to a file, similar to how `cout` prints data to the screen.

`ofstream` is a special kind of stream designed to write data to files. Once a file is opened with an `ofstream` object, you can use:

- `<<` to write text, words, numbers, or variables.
- `\n` or `std::endl` to create new lines.

It works just like `cout`, but instead of going to the screen, the output is written into a file.

To use `ofstream`, include:

```
#include <fstream>   // Provides ofstream
```

❖ 10.3.1. Opening a File with `ofstream`

There are two common ways to open a file for writing:

Method 1: Declare and Open Separately

```
std::ofstream file;         // Create file stream object
file.open("output.txt");    // Open (or create) the file
```

Method 2: Open During Declaration

```
std::ofstream file("output.txt");  // Opens or creates the file immediately
```

Items of interest:

- If the file does not exist, `ofstream` will **create** it.
- If the file does exist, it will be **overwritten** unless you open it in append mode (discussed below).
- You can use a relative path (`"output.txt"`) or an absolute path (`"C:/files/output.txt"`).

❖ 10.3.2. Writing to a File

Below are examples demonstrating different writing techniques:

Example 1: Writing Simple Text Using <<


```
#include <iostream>
#include <fstream>

int main() {
    // Create an ofstream object and try to open "output.txt"
    std::ofstream file("output.txt");

    // Check if the file failed to open
    if (!file) {
        std::cout << "Could not create file.\n";
        return 1;
    }

    // Write text to the file
    file << "Hello, this is a test!" << std::endl;
    file << "The answer is: " << 42 << "\n";

    // Close the file after finishing
    file.close();

    return 0;
}
```

`output.txt` now contains:

```
Hello, this is a test!
The answer is: 42
```

Example 2: Writing Multiple Lines

```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream file("log.txt");

    if (!file) {
        std::cout << "Could not open log file.\n";
        return 1;
    }

    file << "Log Start\n";
    file << "User logged in\n";
    file << "Operation completed\n";

    file.close();

    return 0;
}
```


This is useful for writing logs, saving output, or generating reports.

❖ 10.3.3. Appending to a File (Keep Existing Data)

To add to a file instead of overwriting it, open it in **append mode**:

```
std::ofstream file("log.txt", std::ios::app);  // Append mode
file << "New entry added.\n";
```

Append mode **keeps** old content and adds new data at the end.

❖ 10.3.4. Closing the File

Files can be closed:

```
file.close();   // Write data to disk and free resources
```

C++ closes files processed with ofstream, ifstream, and fstream automatically when the files go out of scope (e.g., the function that opened the file completes). Even so, it is good practice to close files manually.

❖ 10.3.5. Key Points to Remember

- Use **ofstream** for writing to files.
- Include `<fstream>`.
- Always check if the file opened successfully.
- Use `<<` to write data into the file.
- Close the file when done.

- Use `std::ios::app` if you want to append instead of overwrite.

Demo 10.2: FileIo/Demos/ofstream.cpp

```
1.   #include <iostream>   // For printing messages to the screen
2.   #include <fstream>    // Needed for ofstream (output file stream)
3.
4.   int main() {
5.
6.       // Step 1: Create an ofstream object and open the file. If the file does not
               exist, C++ will create it. If it DOES exist, its contents will be
              overwritten.
7.       std::ofstream file("output.txt");   // Opens/creates "output.txt"
8.
9.       // Step 2: Always check if the file opened successfully.
10.      if (!file) {
11.          std::cout << "Error: Could not open file.\n";
12.          return 1;  // Exit the program with an error code
13.      }
14.
15.      // Step 3: Write different types of data to the file. Writing works just
              like cout, but the text goes into the file.
16.      file << "Welcome to the ofstream demo!\n";
17.      file << "This file was created by C++.\n";
18.
19.      int age = 25;
20.      file << "Your age is: " << age << "\n";  // Writing variables
21.
22.      file << "Here are some numbers:\n";
23.      for (int i = 1; i <= 5; ++i) {
24.          file << i << " ";  // Write numbers on the same line
25.      }
26.      file << "\n";  // New line after the loop
27.
28.      // Step 4: Close the file. This saves data and frees system resources.
29.      file.close();
30.
31.      std::cout << "Data written to output.txt successfully!\n";
32.
33.      return 0;
34.  }
```


❖ 10.3.6. File Components after Execution

After running the program, look for `output.txt` in the same folder. It should contain:

```
Welcome to the ofstream demo!
This file was created by C++.
Your age is: 25
Here are some numbers:
1 2 3 4 5
```


10.4. File Modes and Error Handling

When working with files in C++, it's important to know how to open them correctly and how to detect errors if something goes wrong. C++ gives you several file modes that control how a file is opened (read, write, append, etc.), and it provides ways to check whether operations were successful.

❖ 10.4.1. What Are File Modes?

File modes tell C++ *how* to open a file. Some common modes include:

- `std::ios::in` - Open for reading (used by `ifstream`).
- `std::ios::out` - Open for writing (used by `ofstream`).
- `std::ios::app` - Append to the end of the file.
- `std::ios::trunc` - Clear the file when opening (default for `ofstream`).
- `std::ios::binary` - Open in binary mode.

To use file modes and file streams, include:

```
#include <fstream>   // Provides ifstream, ofstream, fstream, and file modes
```

When using the above-mentioned file modes, you combine them with the `|` symbol:

```
std::ios::out | std::ios::app   // Write + Append
```

❖ 10.4.2. Opening Files with Specific Modes

You can open files using fstream, which works for both reading and writing, or with `ifstream`/`ofstream` plus modes.

Example 1: Opening a File in Write + Append Mode

```
#include <iostream>
#include <fstream>

int main() {
    // Open file for writing, but DO NOT erase existing content
    std::ofstream file("log.txt", std::ios::app);

    if (!file) {
        std::cout << "Error: Could not open log.txt\n";
        return 1;
    }

    file << "New log entry.\n";   // Appends to the file
    file.close();
}
```


This keeps old data and adds new content at the end.

Example 2: Opening a File in Read + Write Mode

```
#include <iostream>
#include <fstream>

int main() {
    // fstream allows both reading and writing
    std::fstream file("data.txt",
                      std::ios::in | std::ios::out);

    if (!file) {
        std::cout << "Error: Could not open data.txt\n";
        return 1;
    }

    // Write to the file
    file << "Hello!\n";

    // Move back to the beginning to read again
    file.seekg(0);

    std::string text;
    file >> text;
    std::cout << "First word: " << text << std::endl;

    file.close();
}
```


❖ 10.4.3. Error Handling in File Operations

Always check whether:

1. The file opened successfully.
2. Each read/write operation worked correctly.
3. The program reached the end-of-file (EOF) correctly.

Checking if a File Opened Correctly

```
std::ifstream file("info.txt");

if (!file) {
    std::cout << "Failed to open info.txt\n";
    return 1;
}
```

If the file does not exist or permissions are wrong, `file` becomes *false*.

❖ 10.4.4. Detecting Read/Write Errors

C++ provides several stream-checking functions:

- `fail()` - true if a read/write operation failed.
- `eof()` - true if end of file reached.
- `good()` - true if everything is fine.
- `bad()` - true if a serious error occurred.

Example 3: Safe Reading with Error Checking

```
#include <iostream>
#include <fstream>

int main() {
    std::ifstream file("numbers.txt");

    if (!file) {
        std::cout << "Cannot open numbers.txt\n";
        return 1;
    }

    int value;

    while (file >> value) {     // Read until failure or end of file
        std::cout << "Read: " << value << std::endl;
    }

    if (file.eof()) {
        std::cout << "Reached end of file.\n";
    } else if (file.fail()) {
        std::cout << "Error: Failed to read data.\n";
    }

    file.close();
    return 0;
}
```


❖ 10.4.5. Key Points to Remember

- File modes control *how* files are opened (read, write, append, etc.).
- Use `fstream` when you need both reading and writing.
- Always check if a file opened successfully.
- Use functions like `fail()`, `eof()`, and `good()` to check for errors.

- Close your files to free resources and ensure changes are saved.

Demo 10.3: FileIo/Demos/ModesAndErrorHandling.cpp

```
1.  #include <iostream>
2.  #include <fstream>   // Needed for ifstream, ofstream, and fstream
3.  #include <string>
4.
5.  int main() {
6.
7.      // Part 1: Opening a file in write mode (std::ios::out). This will CREATE
            the file if it doesn't exist. If it DOES exist, its content will be
            ERASED (truncated).
8.      std::ofstream writeFile("example.txt", std::ios::out);
9.
10.     if (!writeFile) {
11.         std::cout << "Error: Could not open example.txt for writing.\n";
12.         return 1;   // Exit if opening failed
13.     }
14.
15.     writeFile << "This is a test line.\n";
16.     writeFile << "Writing to a file using ios::out mode.\n";
17.
18.     writeFile.close();   // Always close the file
19.     std::cout << "Write operation complete.\n\n";
20.
21.     // Part 2: Opening the same file in append mode (std::ios::app). This mode
            does NOT erase existing content. New text is added to the END of the
            file.
22.     std::ofstream appendFile("example.txt", std::ios::app);
23.
24.     if (!appendFile) {
25.         std::cout << "Error: Could not open example.txt for appending.\n";
26.         return 1;
27.     }
28.
29.     appendFile << "This line was added using append mode.\n";
30.
31.     appendFile.close();
32.     std::cout << "Append operation complete.\n\n";
33.
34.     // Part 3: Reading from the file using ifstream. We add error handling and
            end-of-file detection.
35.     std::ifstream readFile("example.txt", std::ios::in);
36.
37.     if (!readFile) {
38.         std::cout << "Error: Could not open example.txt for reading.\n";
39.         return 1;
40.     }
```

Demo 10.3: FileIo/Demos/ModesAndErrorHandling.cpp

```
41.
42.       std::cout << "Contents of example.txt:\n";
43.
44.       std::string line;
45.
46.       // Read the file line by line until EOF is reached
47.       while (std::getline(readFile, line)) {
48.           std::cout << line << std::endl;
49.       }
50.
51.       // Check why the loop ended
52.       if (readFile.eof()) {
53.           std::cout << "\nReached end of file.\n";
54.       } else if (readFile.fail()) {
55.           std::cout << "\nError: A read operation failed.\n";
56.       }
57.
58.       readFile.close();
59.       std::cout << "Read operation complete.\n\n";
60.
61.       // Part 4: Using fstream for both reading and writing. We open with:
                std::ios::in  -> read and std::ios::out -> write. This requires the
                file to already exist.
62.       std::fstream file("example.txt", std::ios::in | std::ios::out);
63.
64.       if (!file) {
65.           std::cout << "Error: Could not open example.txt for read/write.\n";
66.           return 1;
67.       }
68.
69.       // Write a new line at the end
70.       file.clear();               // Clear any EOF flags
71.       file.seekp(0, std::ios::end);  // Move write position to the end
72.       file << "Added using fstream read/write mode.\n";
73.
74.       file.close();
75.       std::cout << "Combined read/write operation complete.\n";
76.
77.       return 0;
78.   }
```

❖ 10.4.6. File Components After Execution

After running the program, your **example.txt** file will contain:

```
This is a test line.
Writing to a file using ios::out mode.
This line was added using append mode.
Added using fstream read/write mode.
```


10.5. Working with Structured Data Files

Structured data files are files that contain records that are composed of **fields**. Fields store individual data items.

For example, consider the following time card data:

Employee Name	Hours Worked	Hourly Wage
Stephen Withrow	40	25.00
Jared Dunn	42	27.00
Cherie Burnett	35	32.00

A data item (e.g., `Employee Name`) is a field. Each record contains time card data that pertains to **an individual employee**. Therefore each row of the table above corresponds to an employee **record**.

Many applications benefit from field-oriented records. To name a few:

- Human Resources (Employee name, hours worked, hourly wage)
- Order Entry (Customer number, Order ID, Product code ...)
- Shipping (Customer address, Billing address, Order ID, Line item number ...)
- Project Management (Project ID, Activity name, Task ID ...)

C++ supports different types of structured data files:

- **CSV** (Comma-separated values) - fields are separated by commas

- **Fixed Length** - fields have a fixed length in the record
- **JSON** (**J**ava**S**cript **O**bject **N**otation) - fields are stored in JSON format

❖ 10.5.1. Working with CSV Files

Comma-separated values (CSV) files are very common in IT data processing. They are a popular means of communicating **spreadsheet** data to C++ programs.

As the name implies, fields are separated from each other by a comma (,). String data is usually enclosed in quotation marks ("). Numeric data is typically **not** enclosed in quotation marks.

We will return to the time card data used in the above table-here are the fields again:

- Employee Name
- Hourly Wage
- Hours Worked

Here is the actual CSV data file:

```
"Stephen Withrow",40,25.00
"Jared Dunn",42,27.00
"Cherie Burnett",35,32.00
"Chris Smith",40,42.00
"Justin Hayward",32,30.00
```

Notice that the fields are separated with a comma and string data is enclosed in quotation marks.

We wish to read the record and extract each field so that we can produce a report that shows net pay for each employee.

Here is the code to open the time card data file in preparation for extracting the fields from each record:

```
#include <iostream>
#include <fstream>   // use for ifstream
#include <sstream>   // use for stringstream to parse fields on one record
#include <string>

int main() {
    std::ifstream timeCard("TimeCard.txt");
    if (!timeCard.is_open()) {
        std::cout << "Cannot open file!";
        return 1;
    }
```

Items of interest:

1. `#include <sstream>` : we will use the `stringstream` class to isolate the individual fields. `sstream` is the header file for the `sstringstream`.
2. We open the time card data file by creating an `ifstream` object..

Now we will read each record and extract the fields:


```
std::string record;
while (std::getline(timeCard, record)) {
    std::stringstream sStream(record);
    std::string employeeName, hoursWorkedStr, hourlyWageStr;
    // get employee name
    std::getline(sStream, employeeName, ','); // get data up to the first comma
    // remove quotation marks from employee name
    employeeName = employeeName.substr(1, employeeName.size() - 2);
    // get hours worked
    std::getline(sStream, hoursWorkedStr, ',');
    // get hourly wage
    std::getline(sStream, hourlyWageStr);
    // convert to numeric
    int hoursWorked = std::stoi(hoursWorkedStr);
    float hourlyWage = std::stof(hourlyWageStr);
    // print output
    std::string reportLine =
        std::format("{} worked {} hours at ${} per hour. Gross pay is ${}.\n",
               employeeName, hoursWorked, hourlyWage, hoursWorked* hourlyWage);
    std::cout << reportLine;
}
```

Note the following:

1. `std::stringstream sStream(record)` : creates an `stringstream` object named `sStream`. You can choose the name that is meaningful to you. The argument (`record`) will be populated with a file record.
2. `std::getline(sStream, employeeName, ',')` : will get the data starting from the beginning of the record up to but not including ',' (comma). The data will be stored in `employeeName`.
3. `std::getline(sStream, hoursWorkedStr, ',')` : will get the data starting after the first comma up to but not including the second comma. The data will be stored in `hoursWorkedStr`.
4. `std::getline(sStream, hourlyWageStr)` : will get the data starting after the second comma up to but not including the first space (the default). The data will be stored in `hoursWageStr`.

Now that the fields have been extracted from the record, we can prepare our report.

First, we have to convert the string forms of hours worked and hourly wage to numeric. These fields are stored as string data on the record, and we cannot do arithmetic on strings.

From the example above, here is the code to handle the conversions from string to numeric:

```
#include <string>
...
int hoursWorked = std::stoi(hoursWorked);
float hourlyWage = std::stof(hourlyWageStr);
```

Note the following:

1. `#include <string>` : header file for the conversion functions.
2. `std::stoi` : string to integer function. Converts the string representation of an integer to `int`.
3. `std::stof` : string to float function. Converts the string representation of a floating-point number to `float`.

Now we can print out the report line. Note that we multiply hours worked times hourly wage to display the gross pay for the employee:

```
std::string reportLine = std::format("{} worked {} hours at ${} per hour. Gross pay is
${}.\n",employeeName, hoursWorked, hourlyWage, hoursWorked* hourlyWage);
std::cout << reportLine;

...
timeCard.close();
```

Output

```
Stephen Withrow worked 40 hours at $25 per hour. Gross pay is $1000.
Jared Dunn worked 42 hours at $27 per hour. Gross pay is $1134.
Cherie Burnett worked 35 hours at $32 per hour. Gross pay is $1120.
Chris Smith worked 40 hours at $42 per hour. Gross pay is $1680.
Justin Hayward worked 32 hours at $30 per hour. Gross pay is $960.
```


When you're finished reading a file, you should **close** the file.

```
timeCard.close();
```

Closing a file is important because the operation frees resources associated with the file handle. In addition, any locks put on the file by the operating system are removed.

You can find the examples above in this demo file:

Demo 10.4: FileIo/Demos/FileReadCSV.cpp

```
1.   #include <iostream>
2.   #include <format>
3.   #include <fstream>   // use for ifstream
4.   #include <sstream>   // use for stringstream to parse fields on one record
5.   #include <string>
6.
7.   int main() {
8.       std::ifstream timeCard("TimeCard.txt");
9.       if (!timeCard.is_open()) {
10.          std::cout << "Cannot open file!";
11.          return 1;
12.      }
13.      std::string record;
14.      while (std::getline(timeCard, record)) {
15.          std::stringstream sStream(record);
16.          std::string employeeName, hoursWorkedStr, hourlyWageStr;
17.          // get employee name
18.          std::getline(sStream, employeeName, ','); // get data up to the first
              comma
19.          // remove quotation marks from employee name
20.          employeeName = employeeName.substr(1, employeeName.size() - 2);
21.          // get hours worked
22.          std::getline(sStream, hoursWorkedStr, ',');
23.          // get hourly wage
24.          std::getline(sStream, hourlyWageStr);
25.          // convert to numeric
26.          int hoursWorked = std::stoi(hoursWorkedStr);
27.          float hourlyWage = std::stof(hourlyWageStr);
28.          // print output
29.         std::string reportLine = std::format("{} worked {} hours at ${} per hour.
               Gross pay is ${}.\n",
30.             employeeName, hoursWorked, hourlyWage, hoursWorked* hourlyWage);
31.          std::cout << reportLine;
32.      }
33.      timeCard.close();
34.  }
```


❖ 10.5.2. Working with Fixed-Length Fields

A file record may consist of **fixed-length fields**, i.e., fields that have a uniform length on the record. The fields are *not* delimited with commas.

Consider the following time card data that is in a fixed form:

```
Stephen Withrow     040025.00
Jared Dunn          042027.00
Cherie Burnett      035032.00
Chris Smith         040042.00
Justin Hayward      032030.00
```


and we are given:

- Employee name is in positions 1-20
- Hours worked is in positions 21-23
- Hourly wage is in positions 24-28

Let's read the file and produce the employee gross pay report:

```
#include <string>   // use for getline
#include <iomanip>  // use to make float data pretty
#include <iostream>
#include <fstream>  // use for ifstream

int main() {
    // read records that have fixed fields:
    // employeeName positions 1:20
    // hoursWorked  positions 21:23
    // hourlyWage   positions 24:28
    std::ifstream timeCardDataFixed("TimeCardDataFixed.txt");


    if (!timeCardDataFixed) {
        std::cout << "Could not open file!";

    }

    std::string record;

    while (std::getline(timeCardDataFixed, record)) {
        std::string employeeName = record.substr(0, 19);
        int hoursWorked = std::stoi(record.substr(20,3));
        float hourlyWage = std::stof(record.substr(23, 5));
        std::string reportLine = std::format("Name: {} Hours Worked: {} Wage: {}\n",
    employeeName, hoursWorked, hourlyWage);
        std::cout << reportLine;
        float netPay = hoursWorked * hourlyWage;
        std::cout << "Net pay: $" << std::fixed << std::setprecision(2) << netPay <<
"\n";
    }


    timeCardDataFixed.close();
}
```

Output

```
Name: Stephen Withrow      Hours Worked: 40 Wage: 25
Net pay: $1000.00
Name: Jared Dunn           Hours Worked: 42 Wage: 27
Net pay: $1134.00
Name: Cherie Burnett       Hours Worked: 35 Wage: 32
Net pay: $1120.00
Name: Chris Smith          Hours Worked: 40 Wage: 42
Net pay: $1680.00
Name: Justin Hayward       Hours Worked: 32 Wage: 30
Net pay: $960.00
```


Items of interest:

1. We use `ifstream` to read the file as we've done in the previous examples.
2. To extract each field from the input record, we use the `substr` function. For example:

```
std::string employeeName = record.substr(0, 20);
```

The employee name is extracted from position `0` (the first position on the input record) for a length of `20`.

Try out these examples using the demo file below:

Demo 10.5: FileIo/Demos/FileReadFixed.cpp

```
1.   #include <iostream>
2.   #include <fstream>  // use for ifstream
3.   #include <sstream>  // use for stringstream
4.   #include <string>   // use for getline
5.   #include <format>
6.   #include <iomanip>  // use to make float data pretty
7.
8.   int main() {
9.       // read records that have fixed fields:
10.      // employeeName positions 1:20
11.      // hoursWorked  positions 21:23
12.      // hourlyWage   positions 24:28
13.      std::ifstream timeCardDataFixed("TimeCardDataFixed.txt");
14.      if (!timeCardDataFixed) {
15.          std::cout << "Could not open file!";
16.      }
17.      std::string record;
18.      while (std::getline(timeCardDataFixed, record)) {
19.          std::string employeeName = record.substr(0, 20);
20.          int hoursWorked = std::stoi(record.substr(20,3));
21.          float hourlyWage = std::stof(record.substr(23, 5));
22.         std::cout << std::format("Name: {} Hours Worked: {} Wage: {}\n", employ  ↵↵
              eeName, hoursWorked, hourlyWage);
23.          float netPay = hoursWorked * hourlyWage;
24.         std::cout << "Net pay: $" << std::fixed << std::setprecision(2) << netPay
               << "\n";
25.      }
26.      timeCardDataFixed.close();
27.      // uncomment the following lines to keep the console open
28.      // for "externalConsole : true" in launch.json (Windows only)
29.      //std::cout << "Press enter to exit...\n";
30.      //std::cin.get();
31.  }
```


❖ 10.5.3. Writing to a File

In order to add records to a file, we must **write** to the file.

Opening a File

We need to open the file before attempting to write to the file. Therefore, we will create an instance of the `ofstream` class:

```
#include <fstream>   // for File I/O operations
...
    std::ofstream timeCardData("TimeCardData.txt");
```

The `ofstream` function header is contained in `<fstream>`. We name the file handle `timeCardData`. As was the case when reading a file, all file operations will be applied using the file handle.

Writing Data

Once the file is open, you can write data to it using the `<<` operator. In this case, we are using hard-coded time card data:

```
timeCardData << "\"Stephen Withrow\",40,25.00\n";
timeCardData << "\"Jared Dunn\",42,27.00\n";
timeCardData << "\"Cherie Burnett\",35,32.00\n";
timeCardData << "\"Chris Smith\",40,42.00\n";
timeCardData << "\"Justin Hayward\",32,30.00\n";
timeCardData.close();
```


Items of interest:

1. Each record is written using the `<<` operator. The file handle `timeCardData` is referenced with the `<<` operator.
2. We use `\"` to **escape** the quotation character (") so that C++ will treat the quotations before and after employee name as data.
3. A **newline character** (`\n`) indicates the **end** of a record (line).
4. The file is closed after all records have been written using the `close` function.

Closing the file is particularly important to ensure that **all records** are written to the file. During the write process, C++ may elect to **buffer records in memory**. Closing the file will cause the buffered records will be stored persistently on disk.

You can run the examples in this section using the demo file below:

Demo 10.6: FileIo/Demos/FileIO.cpp

```
1.   #include <iostream>
2.   #include <fstream>   // for File I/O operations
3.   #include <sstream>   // for getLine
4.
5.   int main() {
6.       // create an ofstream object and load file with hard-coded data:
7.       std::ofstream timeCardData("TimeCardData.txt");   // use this object to
             overwrite exising records
8.       //std::ofstream timeCardData("TimeCardData.txt", std::ios::app);   //
             std::ios::app will append records to file
9.       timeCardData << "\"Stephen Withrow\",40,25.00\n";
10.      timeCardData << "\"Jared Dunn\",42,27.00\n";
11.      timeCardData << "\"Cherie Burnett\",35,32.00\n";
12.      timeCardData << "\"Chris Smith\",40,42.00\n";
13.      timeCardData << "\"Justin Hayward\",32,30.00\n";
14.      timeCardData.close();
15.
16.      // create ifstream object so we can read the file created above
17.      std::ifstream timeCard("TimeCardData.txt");
18.      if (!timeCard.is_open()) {
19.          std::cout << "Cannot open file!";
20.          return 1;
21.      }
22.      std::string record = "";
23.      std::cout << "Printing input file records..." << "\n";
24.      while (std::getline(timeCard, record)) {
25.          std::cout << record << "\n";
26.      }
27.
28.      // we will read the Time Card file again
29.      // and write each record to an output file
30.      timeCard.clear();   // clear the EOF switch
31.      timeCard.seekg(0);  // set g(et) pointer, i.e., rewind to the beginning of
             the file
32.
33.     // open the output file in Input/Output mode and trunc (remove) any existing
            records
34.      std::fstream timeCardOUT("TimeCardOUT.text", std::ios::in | std::ios::out |
             std::ios::trunc);
35.      // use output mode to write records to the file
36.      while (std::getline(timeCard, record)) {
37.          timeCardOUT << record << "\n";
38.      }
39.      timeCardOUT.seekg(0);  // rewind the file
40.      // use input mode to verify records were written
```

Demo 10.6: FileIo/Demos/FileIO.cpp

```
41.       std::cout << "\nPrinting output file records..." << "\n";
42.       while (std::getline(timeCardOUT, record)) {
43.           std::cout << record << "\n";
44.       }
45.       timeCard.close();
46.       timeCardOUT.close();
47.   }
```

❖ 10.5.4. Working With JSON Data

JSON (**J**ava**S**cript **O**bject **N**otation) is a popular format for working with data in C++, as well as other languages such as Python, Java, and JavaScript. The technology is derived from **serializing** JavaScript objects into a string format.

The general form of a JSON file is shown below:

```
[
    { "property1" : value,
      "property2" : value,
      ...
     },
     { "property1" : value,
       "property2" : value,
      ...
     },
     ...
 ]
```


Items of interest:

1. Square brackets (`[ ]`) denote an **array** of JSON objects.
2. Each object (e.g., employee time card data) is enclosed in curly braces (`{}`). We could interpret the properties within the curly braces as a **record**.
3. A field is described as a **property** in JSON. `property1`, for example, might be `EmployeeName`. Property names are enclosed in quotation marks (`"`).
4. The **value** of each property follows the property name and is separated from the property name by a colon (`:`). **String values** are enclosed in quotation marks (`"`).

We can now present employee time card data in JSON format as shown below:

```
[
  {
    "EmployeeName": "Stephen Withrow",
    "HoursWorked": 40,
    "HourlyWage": 25.00
  },
  {
    "EmployeeName": "Jared Dunn",
    "HoursWorked": 42,
    "HourlyWage": 27.00
  },
  {
    "EmployeeName": "Cherie Burnett",
    "HoursWorked": 35,
    "HourlyWage": 32.00
  },
  {
    "EmployeeName": "Chris Smith",
    "HoursWorked": 40,
    "HourlyWage": 42.00
  },
  {
    "EmployeeName": "Justin Hayward",
    "HoursWorked": 32,
    "HourlyWage": 30.00
  }
]
```


The data above is stored in **serialized form**, which is required so that the data is human readable. The data must be **parsed** (converted) into a JSON object before a C++ program can read property values using the property names.

To parse JSON data we will use the Nlohmann JSON library (`https://github.com/nlohmann/json/releases/v3.12.0`). This library is highly regarded in the C++ community for working with JSON.

The example below takes the JSON employee time card data and prints the data in both a raw form and a pretty format:

```
#include "json.hpp"
...

using json = nlohmann::json;  // assign alias for the namespace to simplify coding

int main() {
    std::ifstream timeCard("TimeCard.json");
    ...
    json timeCardJSON;   // define the JSON object

    try {
        timeCard >> timeCardJSON;  // parse(convert) string data from file to JSON
    }
    catch (const json::parse_error& e) {
        std::cerr << "JSON Parse Error: " << e.what() << '\n';
        return 1;
    }

    // Display the JSON file
    std::cout << "Raw JSON Input:\n";
    std::cout << timeCardJSON.dump() << "\n\n";  // dump JSON records to the console

    // Pretty output
    std::cout << std::format("{:^20}{:^15}{:^15}\n", "Employee Name","Hours Worked",
"Hourly Wage");
    for (const auto& tcj : timeCardJSON) {
        std::cout << std::format("{:<20}", tcj["EmployeeName"].get<std::string>())
                  << std::format("{:>15}", tcj["HoursWorked"].get<int>())
                  << std::format("{:>15.2f}\n", tcj["HourlyWage"].get<float>());
    }

    return 0;
}
```


Note the following:

1. `#include "json.hpp"` imports the Nlohmann header file that includes all of the C++ headers, templates, and classes required to process JSON in your C++ program. This file is resident in **the same directory** as the sample program and is not on the include path of C++. Therefore, the header file name is enclosed in **quotation marks** as opposed to angle brackets (`<>`).

2. `using json = nlohmann::json` assigns an **alias** (`json`) to the `json` class within the `nlohmann` namespace.

3. `json timeCardJSON` defines `timeCardJSON` as a **JSON object**.

4. `timeCard >> timeCardJSON`**parses** the incoming data read from the input file into a JSON object.

5. The `json::parse_error` exception will be thrown if the data is not in proper JSON format.

6. The `dump()` function returns the JSON **serialized** data.

7. In the for loop the values of properties for each record is printed. A record is stored in the variable `tcj`.

    A. A property value is accessed using `tcj[property_name]`, e.g., `tcj["EmployeeName"]`.

    B. **Getters** provided in the Nlohmann library convert the parsed JSON field values to `std::string`, `int`, or `float` data types. For example, `tcj["EmployeeName"].get<std::string>()` will return a string object containing the value of `EmployeeName`.

Here is the output of the program:

```
Raw JSON Input:
[{"EmployeeName":"Stephen Withrow","HourlyWage":25.0,"HoursWorked":40},{"Employ ↵↵
eeName":"Jared Dunn","HourlyWage":27.0,"HoursWorked":42},{"EmployeeName":"Cherie Bur ↵↵
nett","HourlyWage":32.0,"HoursWorked":35},{"EmployeeName":"Chris Smith","Hourly ↵↵
Wage":42.0,"HoursWorked":40},{"EmployeeName":"Justin Hayward","HourlyWage":30.0,"Hour ↵↵
sWorked":32}]

   Employee Name     Hours Worked    Hourly Wage
Stephen Withrow                40          25.00
Jared Dunn                     42          27.00
Cherie Burnett                 35          32.00
Chris Smith                    40          42.00
Justin Hayward                 32          30.00
```

Try out the program by running the demo:

Demo 10.7: FileIo/Demos/FileReadJSON.cpp

```
1.   #include <iostream>
2.   #include <fstream>
3.   #include <string>
4.   #include <format>
5.   #include "json.hpp"
6.
7.   using json = nlohmann::json;  // assign alias for the namespace to simplify
          coding
8.
9.   int main() {
10.      std::ifstream timeCard("TimeCard.json");
11.      if (!timeCard.is_open()) {
12.          std::cerr << "Error: Could not open employees.json\n";
13.          return 1;
14.      }
15.
16.      json timeCardJSON;  // define the JSON object
17.
18.      try {
19.          timeCard >> timeCardJSON;  // convert(parse) string data from file to
              JSON
20.      }
21.      catch (json::parse_error& pe) {
22.          std::cerr << "JSON Parse Error: " << pe.what() << '\n';
23.          return 1;
24.      }
25.
26.      // Display the JSON file
27.      std::cout << "Raw JSON Input:\n";
28.      std::cout << timeCardJSON.dump() << "\n\n"; // dump JSON records to the
              console
29.
30.      // Pretty output
31.      std::cout << std::format("{:^20}{:^15}{:^15}\n", "Employee Name","Hours
              Worked", "Hourly Wage");
32.      for (const auto& tcj : timeCardJSON) {
33.         std::cout << std::format("{:<20}", tcj["EmployeeName"].get<std::string>())

34.                   << std::format("{:>15}", tcj["HoursWorked"].get<int>())
35.                  << std::format("{:>15.2f}\n", tcj["HourlyWage"].get<float>());
36.      }
37.
38.      return 0;
39.  }
```

Exercise 10: Practice File I/O

10 to 20 minutes

In this exercise you will add file I/O to the Banking Menu app.

❖ E10.1. Requirements

1. The account information will be written to a text file.
2. The logs of deposits and withdrawals will also be written to the text file.
3. The user will be given the option of viewing the contents of the file after entering all withdrawals and deposits.

❖ E10.2. Step-by-step Instructions

1. Open `FileIO\Exercises\BankingMenu.cpp` in your editor as your starter file.
2. After the user has exited the `do...while` loop, open `AccountLog.txt` for output.
3. Write the account information to the output file using the `display` function of the `account` object. **Note**: previously you printed the results of the `display` function to the console.
4. Write the log of withdrawals to the file instead of printing the log to the console.
5. Write the log of deposits to the file instead of printing the log to the console.
6. Close the log file.
7. Ask the user if he or she wishes to view the account log file.
8. If the user replies 'Y' or 'y' then open `AccountLog.txt` for input and display all records to the console. Close the file.
9. If user does not wish to view the output records then no other statements need to be run.

Solution: FileIo/Solutions/BankingMenu.cpp

```
1.  #include <iostream>
2.  #include <format>
3.  #include <stdexcept>
4.  #include <string>
5.  #include <vector>
6.  #include <map>
7.  #include <algorithm>
8.  #include <deque>
9.  #include <fstream>
10. #include <memory>
11.
12. typedef std::string str;
13.
14. const double START_BALANCE = 0.0;
15.
16. str userInput;
17.
18. class Account {
19.
20. private:
21.     int acctNbr;
22.     str holder;
23.     double balance;
24.
25. protected:
26.     void setBalance(double balance) { this->balance = balance; }
27.
28. public:
29.     Account(int acctNbr, str holder, double balance)
30.         : acctNbr(acctNbr), holder(holder), balance(balance) {
31.     }
32.     // default constructor (no parameters)
33.     Account() {
34.         acctNbr = 999999999;
35.         holder = "N/A";
36.         balance = 0.0;
37.     }
38.     // Getters
39.     int getAcctNbr() { return acctNbr; }
40.     str getHolder() { return holder; }
41.     double getBalance() { return balance; }
42.
43.     // Setters
44.     void setAcctNbr(int acctNbr) { this->acctNbr = acctNbr; }
```

Solution: FileIo/Solutions/BankingMenu.cpp

```
45.      void setHolder(str holder) { this->holder = holder; }
46.
47.      // virtual display function
48.      virtual std::string display() {
49.          return std::format("Account number {} with holder {} and balance of
              ${:.2f}.\n",
50.             getAcctNbr(), getHolder(), getBalance());
51.      }
52.
53.      //pure virtual functions
54.      virtual void deposit(double deposit) = 0;
55.
56.      virtual void withdraw(double withdrawal) = 0;
57.
58.  };
59.
60.  class Checking : public Account {
61.  private:
62.      bool paperChecks;
63.  public:
64.      Checking(int acctNbr, str holder, double balance, bool paperChecks)
65.          : Account(acctNbr, holder, balance), paperChecks(paperChecks) {
66.      }
67.      void deposit(double amount) override {
68.          if (amount <= 0) {
69.              std::string error_msg =
70.                  std::format("Deposit rejected: amount ${:.2f} is less than or
               equal to zero.\n", amount);
71.              throw std::runtime_error(error_msg);
72.          }
73.          double balance = getBalance();
74.          setBalance(balance + amount);
75.      }
76.      void withdraw(double amount) override {
77.          if (amount <= 0) {
78.              std::string error_msg =
79.                  std::format("Withdrawal rejected: amount ${:.2f} is less than
               or equal to zero.\n", amount);
80.              throw std::runtime_error(error_msg);
81.          }
82.          double balance = Account::getBalance();
83.          Account::setBalance(balance - amount);
84.      }
85.      // virtual display function
86.      virtual std::string display() {
```

Solution: FileIo/Solutions/BankingMenu.cpp

```
87.           return Account::display() + std::format("Paper checks for this Checking
                account is {}.\n", paperChecks);
88.       }
89.   };
90.
91.   class Savings : public Account {
92.   private:
93.       double transactionFee;
94.       // track number of withdrawals for each Savings object:
95.       int nbrOfWithdrawals = 0;
96.   public:
97.       // Transaction fee for a Savings withdrawal
98.       const double SAVINGS_TRANSACTION_FEE = 2.50;
99.       // Fee for exceeding maximum number of Savings withdrawals:
100.      const double SAVINGS_MAX_WD_FEE = 3.00;
101.      // Maximum number of savings withdrawals without fee:
102.      const int MAX_SAVINGS_WITHDRAWALS = 3;
103.
104.      // Constructor with transaction fee
105.      Savings(int acctNbr, str holder, double balance, double transactionFee)
106.          : Account(acctNbr, holder, balance), transactionFee(transactionFee) {
107.      }
108.      // Constructor without transaction fee...default transaction fee will be
            used
109.      Savings(int acctNbr, str holder, double balance)
110.          : Account(acctNbr, holder, balance) {
111.              transactionFee = SAVINGS_TRANSACTION_FEE;
112.      }
113.
114.      void deposit(double amount) override {
115.          if (amount <= 0) {
116.              std::string error_msg =
117.                  std::format("Deposit rejected: amount ${:.2f} is less than or
              equal to zero.\n", amount);
118.              throw std::runtime_error(error_msg);
119.          }
120.          double balance = Account::getBalance();
121.          setBalance(balance + amount);
122.      }
123.      void withdraw(double amount) override {
124.          if (amount > Account::getBalance()) {
125.              throw std::runtime_error("Withdrawal amount exceeds balance");
126.          }
127.          else if (amount <= 0) {
128.              std::string error_msg =
```

Solution: FileIo/Solutions/BankingMenu.cpp

```
129.                 std::format("Withdrawal rejected: amount ${:.2f} is less than
             or equal to zero.\n", amount);
130.             throw std::runtime_error(error_msg);
131.         }
132.         else {
133.             nbrOfWithdrawals++;
134.             if (nbrOfWithdrawals > MAX_SAVINGS_WITHDRAWALS) {
135.                 amount += SAVINGS_MAX_WD_FEE;
136.             }
137.     }
138.         double balance = Account::getBalance() - transactionFee;
139.         setBalance(balance - amount);
140.     }
141.     // virtual display function
142.     virtual std::string display() {
143.        return Account::display() + std::format("Transaction fee for this Savings
              account is ${:.2f}.\n", transactionFee);
144.     }
145. };
146.
147. int main() {
148.
149.     std::vector<double> depositLog{}; // use vector for deposits
150.     std::deque<double> withdrawDeque; // use deque for withdrawals
151.     std::map<std::string, double> transactionsMap;
152.     // set total deposits to 0
153.     transactionsMap["Deposits"] = 0;
154.     // set total withdrawals to 0
155.     transactionsMap["Withdrawals"] = 0;
156.
157.     // prompt user for account holder name:
158.     std::cout << "Please enter account holder name: ";
159.     getline(std::cin, userInput);
160.     std::string accountHolder = userInput;
161.     // create Account object pointer:
162.     std::unique_ptr<Account> account;
163.
164.     bool correct = false;
165.     // prompt the user for account type:
166.     while (!correct && userInput[0] != 'Q') {
167.         bool continueLooping = true; // used in the 'S' case of the switch
              statement
168.         std::cout << "Enter account type (C for Checking, S for Savings) or Q
              to Quit the app:\n";
169.         getline(std::cin, userInput);
```

Solution: FileIo/Solutions/BankingMenu.cpp

```
170.          switch (userInput[0]) {
171.          case 'C':
172.          case 'c':
173.              std::cout << "Do you want paper checks (Y/N)? ";
174.              getline(std::cin, userInput);
175.              if (userInput[0] == 'Y' || userInput[0] == 'y') {
176.                  account = std::make_unique<Checking>(555666777, accountHolder,
              START_BALANCE,  true);
177.              }
178.              else {
179.                  account = std::make_unique<Checking>(555666777, accountHolder,
              START_BALANCE, false);
180.              }
181.              correct = true;
182.              break;
183.          case 'S':
184.          case 's':
185.               do {
186.                  std::cout << "Enter the transaction fee for each withdrawal or
              \'d\' for the default fee ($2.50): ";
187.                  try {
188.                      getline(std::cin, userInput);
189.                      if (userInput[0] == 'd' || userInput[0] == 'D') {
190.                          // user wants to use the default fee
191.                        account = std::make_unique<Savings>(555666777, accoun ↵↵
              tHolder, START_BALANCE);
192.                      }
193.                      else {
194.                          double userTransactionFee = std::stod(userInput);  //
              get user transaction fee
195.                          if (userTransactionFee < 0) {
196.                              // throw invalid argument exception with message
              place holder
197.                              // (hard coded message will be displayed in catch
              block)
198.                              throw std::invalid_argument("message");
199.                          }
200.                        account = std::make_unique<Savings>(555666777, accoun ↵↵
              tHolder, START_BALANCE, userTransactionFee);
201.                      }
202.                      continueLooping = false;
203.                  }
204.                  catch (std::invalid_argument& ie) {
205.                      std::cout << "You entered a non-numeric or negative fee.
              Please try again.\n";
206.                  }
```

Solution: FileIo/Solutions/BankingMenu.cpp

```
            }
            while (continueLooping);
            correct = true;
            break;
        case 'Q':
        case 'q':
            std::cout << "Closing the app by user request.\n";
            return 0;
        default:
           std::cout << "Unknown account type...please re-enter (Q to quit).\n";
        }
    }

    do {
        // Display menu options
        std::cout << " Deposit (d)\n";
        std::cout << " Withdraw (w)\n";
        std::cout << " Balance (b)\n";
        std::cout << " Quit (q)\n\n";
        std::cout << "Enter choice: ";
        std::getline(std::cin, userInput);
        switch (userInput[0]) {
            // Handle deposit
            case 'd':
            case 'D': {
                double amount;
                std::cout << "Enter amount to deposit: ";
                std::getline(std::cin, userInput);
                try {
                    amount = std::stod(userInput);
                    account->deposit(amount);
                    // log deposit amount to vector
                    depositLog.push_back(amount);
                    // add deposit amount to running total on the map
                    transactionsMap["Deposits"] += amount;
                    std::cout << std::format("Deposited ${:.2f}\n", amount);
                }
                catch (std::invalid_argument& ie) {
                   std::cout <<  "Deposit amount must be numeric. Please try
            again.\n";
                }
                catch (std::runtime_error& re) {
                   std::cout << re.what();
                }
```

Solution: FileIo/Solutions/BankingMenu.cpp

```
250.                 break;
251.             }
252.             // Handle withdrawal
253.             case 'w':
254.             case 'W': {
255.                 double amount;
256.                 std::cout << "Enter withdrawal amount: ";
257.                 std::getline(std::cin, userInput);
258.                 try {
259.                     amount = std::stod(userInput);
260.                     account->withdraw(amount);
261.                     // log withdrawal to deque
262.                     // use push_front so that most recent withdrawal is front
            of the deque
263.                     withdrawDeque.push_front(amount);
264.                     // add withdrawal amount to running total on the map
265.                     transactionsMap["Withdrawals"] += amount;
266.                     std::cout << std::format("Withdrew ${:.2f}\n", amount);
267.                 }
268.                 catch (std::invalid_argument& ie) {
269.                  std::cout <<  "Withdrawal amount must be numeric, amount re  ↵↵
            jected.\n";
270.                 }
271.                 catch (std::runtime_error rte) {
272.                     std::cout << rte.what();
273.                 }
274.                 break;
275.             }
276.             // Display balance
277.             case 'b':
278.             case 'B':
279.                 std::cout << std::format("Current balance: ${:.2f}\n", account-
            >getBalance());
280.                 break;
281.             // Handle quit
282.             case 'q':
283.             case 'Q':
284.                 std::cout << std::format("Final balance: ${:.2f}\n", account-
            >getBalance());
285.                 break;
286.             // Handle invalid input
287.             default:
288.               std::cout << "Invalid selection. Please choose a valid option.\n";
289.         }
290.     } while (userInput[0] != 'q' && userInput[0] != 'Q');
```

Solution: FileIo/Solutions/BankingMenu.cpp

```
291.
292.      // open Account Log file
293.      std::ofstream accountLog("AccountLog.txt");
294.      // write account information to account file
295.      accountLog << std::format("{}", account->display());
296.      accountLog << std::format("Total amount of deposits is ${:.2f}.\n", transac ↵↵
              tionsMap["Deposits"]);
297.      accountLog << std::format("Total amount of withdrawals is ${:.2f}.\n",
              transactionsMap["Withdrawals"]);
298.      // write withdrawal log to a file - treat deque as a stack
299.      accountLog << std::format("{:^20}\n", "Log of Withdrawals - Most Recent
              First");
300.      for (const auto& wd : withdrawDeque) {
301.          accountLog << std::format("{:>20}\n", std::format("${:.2f}", wd));
302.      }
303.      // print withdrawDeque as a queue so least recent withdrawals are displayed
              first
304.      accountLog << std::format("{:^20}\n", "Log of Withdrawals-Least Recent
              First");
305.      for (auto wdIter = withdrawDeque.rbegin(); wdIter != withdrawDeque.rend();
              wdIter++) {
306.          accountLog << std::format("{:>20}\n", std::format("${:.2f}", *wdIter));
307.      }
308.      // write deposit log to a file
309.      accountLog << std::format("{:^20}\n", "Log of Deposits");
310.      std::sort(depositLog.begin(), depositLog.end(), std::greater<>{});  // sort
              deposits in descending sequence
311.      for (auto& deposit : depositLog) {
312.          if (deposit == 0) {
313.              break;
314.          }
315.          accountLog << std::format("{:>20}\n", std::format("${:.2f}", deposit));
316.      }
317.      accountLog.close();
318.      std::cout << "Account log available in AccountLog.txt.\n";
319.      std::cout << "Do you want to view the account log?\n";
320.      getline(std::cin, userInput);
321.      if (userInput[0] == 'Y' || userInput[0] == 'y') {
322.          std::ifstream accountLog("AccountLog.txt");
323.          std::string record;
324.          if (accountLog.is_open()) {
325.              while (getline(accountLog, record)) {
326.                  std::cout << record << "\n";
327.              }
328.          }
```

Solution: FileIo/Solutions/BankingMenu.cpp

```
329.         else {
330.             std::cout << "Could not open the account log file.\n";
331.         }
332.         accountLog.close();
333.     }
334. }
```

Conclusion

In this lesson, you learned how to use C++ file streams to read from and write to files by including `<fstream>` and working with `ifstream`, `ofstream`, and `fstream`. You practiced the open use close pattern, wrote with `<<`, read with `>>` and `getline()`, chose appropriate file modes like `std::ios::in`, `std::ios::out`, `std::ios::app`, and handled errors using stream states such as `good()`, `fail()`, `bad()`, and `eof()`.

We explained how to append without overwriting, how to loop until end-of-file safely, and how to verify that files opened successfully. You then applied these skills in the Banking Menu app by writing account details and transaction logs to a text file and optionally reading the log back to the console. You are now prepared to persist, inspect, and manage data reliably using robust file I/O techniques.