C++ Syntax

readbytes

C++ Syntax

Guide to Writing Clear and Correct Code

readbytes.github.io

2025-07-23

This page is intentionally left blank.

Contents

Chapter 1.
Source File and Basic Program Structure

1. C++ Source File Syntax and Structure
2. The `main()` Function
3. Statements and Blocks `{}`
4. Comments: Single-line and Multi-line

1 Source File and Basic Program Structure

1.1 C++ Source File Syntax and Structure

A C++ source file is where you write the code that tells the computer what to do. Understanding the basic syntax rules and structure of a source file is essential for writing clear and correct programs.

1.1.1 What is a C++ Source File?

A C++ source file typically has the extension `.cpp`. It contains a sequence of instructions, declarations, and definitions that make up your program. The compiler reads this file to generate an executable program.

1.1.2 Basic Syntax Rules

- **Headers:** At the top of your source file, you often include *header files* using the `#include` directive. These headers bring in declarations and functionalities from libraries or other files. For example, including `<iostream>` allows your program to use input/output streams like `std::cout`.
- **Namespaces:** Namespaces help organize code and prevent name conflicts. The standard C++ library is contained within the `std` namespace. You can access library features by prefixing them with `std::`, or bring them into your code with `using namespace std;`.
- **Statements:** The instructions you write in C++ are called *statements*. Each statement ends with a semicolon `;`.
- **Blocks:** Code statements can be grouped inside curly braces `{}` to form blocks, which define a scope for variables and group multiple statements together.

1.1.3 How is a C++ Program Structured?

At its simplest, a C++ program must have one special function called `main()`. This is the entry point where execution begins. The `main()` function contains statements that form the body of your program.

Other parts commonly found in source files:

- **Preprocessor directives:** Like `#include` or `#define`.
- **Function declarations and definitions:** Code reusable blocks.
- **Variable declarations and definitions.**

1.1.4 Minimal Valid C++ Program Example

Here is an example of the smallest valid C++ program file:

Full runnable code:

```cpp
#include <iostream>      // Include header for input/output

int main() {             // Main function: program entry point
    std::cout << "Hello, world!\n";  // Print a message to the console
    return 0;            // Return 0 to indicate successful execution
}
```

Let's break down this example:

- `#include <iostream>` tells the compiler to include the input/output stream library.
- `int main()` defines the main function that returns an integer.
- The curly braces `{}` define the start and end of the function body.
- Inside the body, `std::cout << "Hello, world!\n";` sends the text to the console.
- `return 0;` indicates the program finished successfully.

1.1.5 Summary

- A C++ source file contains code organized into statements, blocks, and functions.
- Headers bring in external code with `#include`.
- Namespaces organize code and avoid naming conflicts.
- Every program needs a `main()` function where execution starts.
- Statements end with a semicolon `;`.
- Code blocks are enclosed in `{}`.

This fundamental structure allows you to write clear, organized, and correct C++ programs.

1.2 The `main()` Function

Every C++ program must have a **`main()` function**. This function is the *entry point* of the program — the place where execution begins. When you run a program, the operating system calls the `main()` function first and starts executing the statements inside it.

1.2.1 Purpose of the `main()` Function

- Acts as the starting point for program execution.
- Organizes the overall flow of the program.
- Returns an integer value to the operating system indicating how the program ended.

Without a `main()` function, a C++ program cannot run.

1.2.2 Signatures of `main()`

There are two common valid ways to define `main()`:

1. **No parameters:**
   ```
   int main()
   ```
2. **With command-line parameters:**
   ```
   int main(int argc, char* argv[])
   ```

Both forms return an `int`, representing the program's exit status.

1.2.3 Return Type and Return Value

- The return type of `main()` is always `int`. This is a convention that tells the operating system how the program ended.
- Returning `0` means the program finished successfully.
- Returning a non-zero value indicates an error or abnormal termination.

For example:

```
int main() {
    // program code here
    return 0; // success
}
```

If you omit `return 0;` in `main()` (in C++11 and later), the compiler implicitly adds it, meaning the program ends successfully by default.

1.2.4 Command-Line Parameters: `argc` and `argv`

When a program is run from a command line or terminal, it can receive input parameters from the user. These parameters are passed to `main()` via:

- **`argc` (argument count):** An integer representing the number of command-line arguments passed to the program. This count always includes the program's name itself as the first argument.

- **`argv` (argument vector):** An array of C-style strings (`char*[]`), each representing one command-line argument. The first element `argv[0]` is the program's name or path, and subsequent elements are the arguments.

1.2.5 Example: Simple `main()` Without Parameters

Full runnable code:

```
#include <iostream>

int main() {
    std::cout << "Hello from main!\n";
    return 0;
}
```

1.2.6 Example: `main()` With Command-Line Arguments

Full runnable code:

```
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout << "Program name: " << argv[0] << "\n";
    std::cout << "Number of arguments: " << argc << "\n";

    for (int i = 1; i < argc; ++i) {
        std::cout << "Argument " << i << ": " << argv[i] << "\n";
    }

    return 0;
}
```

If this program is run as:

```
./myprogram apple banana
```

It will output:

```
Program name: ./myprogram
Number of arguments: 3
Argument 1: apple
Argument 2: banana
```

1.2.7 Summary

- `main()` is the mandatory starting point of any C++ program.
- It returns an `int` indicating success (`0`) or failure (non-zero).
- It can have no parameters or accept command-line arguments via `argc` and `argv`.
- Command-line parameters allow users to influence program behavior without changing code.

Understanding `main()` is key to writing functional and interactive C++ programs.

1.3 Statements and Blocks {}

In C++, a **statement** is the basic unit of execution—an instruction that tells the computer to perform a specific action. Understanding statements and how to group them using **blocks** `{}` is essential for writing clear and organized code.

1.3.1 What Is a Statement?

A statement in C++ can take various forms. Here are the most common types:

- **Expression statements:** These perform calculations, assignments, or function calls. They always end with a semicolon `;`.

```
int x = 5;              // Declaration and assignment (expression statement)
x = x + 2;              // Expression modifying x
std::cout << x << "\n";  // Function call statement
```

- **Control flow statements:** These control the flow of execution. Examples include:
  - `if` / `else`
  - `for`, `while`, `do-while` loops
  - `switch` statements
  - `break`, `continue`, `return` statements

Each control flow statement may control one or more statements, often grouped inside **blocks**.

1.3.2 What Is a Block {}?

A **block** is a group of zero or more statements enclosed in curly braces `{}`. Blocks serve two important purposes:

1. **Grouping multiple statements** so that they can be treated as a single statement by

control flow constructs.
2. **Creating a scope** for variables declared inside the block, meaning those variables only exist within the block and cannot be accessed outside it.

1.3.3 Why Use Blocks?

Without blocks, control flow statements like `if` or `for` only apply to the very next single statement. Using blocks allows you to execute multiple statements conditionally or repeatedly.

1.3.4 Example: Single Statement vs. Block

```
if (true)
    std::cout << "Hello\n";   // Applies only to this one statement

if (true) {
    std::cout << "Hello\n";   // Multiple statements grouped in a block
    std::cout << "World!\n";
}
```

1.3.5 Example: Nested Blocks and Scope

Full runnable code:

```
#include <iostream>

int main() {
    int x = 10;  // x is declared in the main function scope

    {
        int y = 20;  // y is declared inside this inner block
        std::cout << "Inside block: x = " << x << ", y = " << y << "\n";
    }

    // y is not accessible here - it only exists inside the inner block
    // std::cout << y;  // ERROR: y is out of scope

    std::cout << "Outside block: x = " << x << "\n";

    return 0;
}
```

Output:

```
Inside block: x = 10, y = 20
```

```
Outside block: x = 10
```

1.3.6 Key Points About Scope

- Variables declared inside a block `{}` are **local to that block**.
- Once the block ends, those variables are destroyed and cannot be used.
- Blocks can be nested inside other blocks to create deeper levels of scope.
- This allows careful management of variable lifetime and helps avoid naming conflicts.

1.3.7 Summary

- A **statement** is a single instruction ending with a semicolon, or a control flow statement controlling one or more statements.
- **Blocks `{}`** group multiple statements, allowing them to be executed together.
- Blocks create **scope**, limiting where variables exist and can be accessed.
- Using blocks and understanding scope improves code clarity, maintainability, and safety.

1.4 Comments: Single-line and Multi-line

Comments are an essential part of writing clear and maintainable C++ code. They allow you to add notes, explanations, or temporarily disable parts of your code without affecting the program's behavior.

1.4.1 What Are Comments?

Comments are pieces of text ignored by the compiler. They are intended for humans reading the code — whether it's yourself, your teammates, or future maintainers. Well-written comments improve code readability and help explain *why* something is done, not just *what* is done.

1.4.2 Two Styles of Comments in C

C++ supports two main styles of comments:

Single-line Comments (//)

- Begin with two forward slashes //.
- Everything following // on that line is ignored by the compiler.
- Useful for brief comments or commenting out a single line.

Example:

```
int x = 5;  // Initialize x with 5
// std::cout << "This line is commented out and won't run\n";
```

Multi-line Comments (/* ... */)

- Begin with /* and end with */.
- Can span multiple lines.
- Useful for longer explanations or temporarily disabling blocks of code.

Example:

```
/*
This function calculates the area of a rectangle.
It takes width and height as input parameters.
*/
int area(int width, int height) {
    return width * height;
}

/*
The following code is disabled for now:
std::cout << "Debug info\n";
*/
```

1.4.3 Using Comments Effectively

- **Document intent:** Explain why the code does something, not just what it does.
- **Keep comments updated:** Outdated comments can confuse more than they help.
- **Avoid obvious comments:** Don't restate what the code clearly does.
- **Use comments to temporarily disable code:** Useful during debugging or testing.

1.4.4 Common Pitfalls with Comments

- **Unclosed multi-line comments:** Forgetting the closing */ causes compilation errors because the compiler treats everything after as a comment.

  ```
  /* This comment is not closed
  int x = 5;  // This line will cause an error
  ```

- **Nested multi-line comments:** C++ does **not** support nesting multi-line comments.

Placing `/* ... /* ... */ ... */` inside another multi-line comment causes issues.

- **Overusing comments:** Too many comments clutter code; strive for self-explanatory code with well-placed comments instead.

1.4.5 Summary

- Use `//` for short, single-line comments.
- Use `/* ... */` for longer, multi-line comments or to disable blocks of code.
- Comments improve readability, explain intent, and help with debugging.
- Be careful with multi-line comment syntax to avoid errors.

Chapter 2.
Primitive Types, Constants, and Data Types

1. Primitive Types: `int`, `float`, `double`, `char`, `bool`
2. `const` and `constexpr`
3. Type Modifiers: `signed`, `unsigned`, `short`, `long`
4. Type Aliases with `typedef` and `using`

2 Primitive Types, Constants, and Data Types

2.1 Primitive Types: `int`, `float`, `double`, `char`, `bool`

Primitive types are the fundamental built-in data types in C++ used to store simple values such as numbers, characters, and boolean flags. Understanding these types is essential for writing effective C++ programs.

2.1.1 Overview of Fundamental Primitive Types

Type	Typical Size (bytes)	Range (approximate)	Usage Scenario
`int`	4	-2,147,483,648 to 2,147,483,647	Whole numbers without fractional parts
`float`	4	±1.2e-38 to ±3.4e+38 (6-7 decimal digits precision)	Single-precision floating-point numbers (decimals)
`double`	8	±2.3e-308 to ±1.7e+308 (15-16 decimal digits precision)	Double-precision floating-point numbers (more accurate decimals)
`char`	1	-128 to 127 or 0 to 255 (depends on signedness)	Single characters or small integers
`bool`	1 (usually)	`true` or `false`	Boolean logic values (true/false flags)

2.1.2 Integer Types: `int`

- Used for whole numbers without decimals.
- Can store positive, negative, and zero values.
- Typically 4 bytes in modern systems, but size can vary by platform.
- Commonly used in counting, indexing, and integer arithmetic.

Example:

```cpp
int age = 30;
int score = -100;
int sum = age + score;
```

2.1.3 Floating-Point Types: `float` and `double`

- Used for numbers with fractional parts (decimals).

- `float` is single precision, suitable when memory is limited or less precision is acceptable.
- `double` provides more precision and is the default choice for floating-point numbers in C++.

Example:

```cpp
float temperature = 36.6f;   // Note the suffix 'f' to specify float literal
double pi = 3.1415926535;
double area = pi * 2.0 * 2.0;
```

2.1.4 Difference Between Integer and Floating-Point Types

- **Integer types** store exact whole numbers.
- **Floating-point types** store approximations of real numbers and can represent fractional values.
- Floating-point arithmetic can introduce rounding errors due to limited precision.

2.1.5 Character Type: `char`

- Stores a single character (letter, digit, symbol).
- Internally stored as a small integer representing the character's code (usually ASCII).
- Can be manipulated as a numeric value or as a character.

Example:

```cpp
char grade = 'A';
char newline = '\n';   // Escape sequence for new line
int charCode = grade;  // Implicit conversion from char to int
```

2.1.6 Boolean Type: `bool`

- Represents truth values: `true` or `false`.
- Used for logical conditions and control flow.
- Underlying implementation stores `true` as 1 and `false` as 0.

Example:

```cpp
bool isPassed = true;
bool hasFinished = false;

if (isPassed) {
    std::cout << "Congratulations!\n";
}
```

2.1.7 Summary

- `int`: whole numbers without decimals.
- `float` and `double`: numbers with decimals; `double` offers more precision.
- `char`: stores single characters, internally numeric.
- `bool`: logical values representing true or false.
- Choose the type based on the kind of data you need to store and precision requirements.

2.2 `const` and `constexpr`

In C++, managing constants properly is key to writing safe, readable, and optimized code. Two important keywords related to constants are `const` and `constexpr`. While both enforce immutability (preventing modification), they serve different purposes and have distinct behaviors, especially regarding **compile-time evaluation**.

2.2.1 What Is `const`?

- The `const` keyword declares a variable whose value **cannot be changed after initialization**.
- The value of a `const` variable is fixed once assigned and cannot be modified.
- However, the initialization of a `const` variable may occur at **runtime**, meaning the value might not be known during compilation.
- `const` is mainly about **immutability**—ensuring variables are read-only.

Example:

```cpp
const int maxUsers = 100;
const double pi = 3.14159;
```

In the example above, `maxUsers` and `pi` cannot be changed later in the program.

2.2.2 What Is `constexpr`?

- Introduced in C++11, `constexpr` variables are **constant expressions** that are evaluated **at compile time**.
- This means their values must be known and computable during compilation.
- `constexpr` implies `const`, so these variables are also immutable.
- Because values are known at compile time, `constexpr` enables compiler optimizations such as replacing variables directly with their values.

Example:

```cpp
constexpr int maxConnections = 500;
constexpr double gravity = 9.81;

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

constexpr int maxSquare = square(10);  // Computed at compile time
```

2.2.3 Key Differences Between `const` and `constexpr`

Feature	`const`	`constexpr`
Immutability	Yes	Yes
Compile-time evaluation	Not required	Required
Initialization	Can be runtime or compile-time	Must be compile-time constant
Usage	Read-only variables	Compile-time constants, constant expressions
Optimizations	Limited	Enables better compile-time optimization

2.2.4 When to Use `const` vs `constexpr`

- Use `const` when the value is **known at runtime** or when you want to make variables read-only but their value might depend on runtime input.
- Use `constexpr` when the value is **known at compile time** or can be computed at compile time. This helps with performance by enabling the compiler to replace variables with literal values and optimize code.

2.2.5 Practical Examples

```cpp
#include <iostream>

const int runtimeValue = std::rand() % 100;   // Value assigned at runtime, but immutable after

constexpr int compileTimeValue = 42;           // Known at compile time
constexpr int factorial(int n) {
    return (n <= 1) ? 1 : (n * factorial(n - 1));
}

constexpr int fact5 = factorial(5);            // Computed at compile time

int main() {
    std::cout << "Runtime const: " << runtimeValue << "\n";
    std::cout << "Compile-time const: " << compileTimeValue << "\n";
    std::cout << "Factorial computed at compile time: " << fact5 << "\n";
    return 0;
}
```

2.2.6 Summary

- Both `const` and `constexpr` create **immutable variables**.
- `const` values can be assigned at runtime; `constexpr` values must be known at compile time.
- Prefer `constexpr` when you want compile-time constants and better optimization.
- Use `const` when immutability is needed but compile-time evaluation isn't possible.

By understanding and applying these correctly, you can write safer, clearer, and more efficient C++ programs.

2.3 Type Modifiers: `signed`, `unsigned`, `short`, `long`

In C++, **type modifiers** are used to alter the size, range, and interpretation of built-in integer types. By applying these modifiers to `int` or other integer types, you can tailor variable behavior to fit specific needs, such as memory optimization, representing only positive numbers, or accommodating larger values.

2.3.1 What Are Type Modifiers?

The most common integer type modifiers are:

- **`signed`**: Explicitly specifies that the integer can hold both positive and negative values.
- **`unsigned`**: Specifies the integer can only represent non-negative values (zero and positive).
- **`short`**: Requests a smaller-than-default integer size.
- **`long`**: Requests a larger-than-default integer size.
- **`long long`**: (C++11) An even larger integer type for very big values.

By default, `int` is signed unless explicitly declared otherwise.

2.3.2 Effects of Modifiers on Size and Range

Type	Typical Size (bytes)	Approximate Range
`signed int`	4	-2,147,483,648 to 2,147,483,647
`unsigned int`	4	0 to 4,294,967,295
`short int` / `short`	2	-32,768 to 32,767
`unsigned short`	2	0 to 65,535
`long int` / `long`	4 or 8 (platform dependent)	-2,147,483,648 to 2,147,483,647 (4 bytes) or larger
`unsigned long`	4 or 8	0 to 4,294,967,295 (4 bytes) or larger
`long long int`	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned long long`	8	0 to 18,446,744,073,709,551,615

2.3.3 Signed vs Unsigned

- **Signed integers** can represent negative numbers, zero, and positive numbers.
- **Unsigned integers** represent only zero and positive numbers but can store a larger positive range compared to signed integers of the same size.

Example:

```
signed int temperature = -20;
unsigned int positiveNumber = 300;
```

Trying to assign a negative number to an `unsigned` variable leads to unexpected results due to underflow.

2.3.4 Short and Long Modifiers

- Use `short` when you want to save memory and the values fit in a smaller range.
- Use `long` or `long long` when you need to store very large integer values.

Example:

```
short int smallNumber = 32000;
long int largeNumber = 1000000000;
long long int veryLargeNumber = 9000000000000000000;
```

2.3.5 When and Why Use Type Modifiers?

- **Portability:** Different systems may have different default sizes for `int`, so using modifiers like `short` and `long` helps write code that better matches the target platform.
- **Memory optimization:** Use smaller types (`short`, `unsigned`) when memory is limited, such as in embedded systems.
- **Correctness and clarity:** Use `unsigned` when negative numbers don't make sense (e.g., counting objects, array indices).
- **Range requirements:** Use `long` or `long long` when values may exceed the range of a standard `int`.

2.3.6 Practical Examples

Full runnable code:

```
#include <iostream>

int main() {
    unsigned int positiveCount = 100;  // Only non-negative values
    short int temperature = -10;       // Smaller storage for a limited range
    long int distance = 1000000;       // Larger range for big numbers
    long long int starsInGalaxy = 1000000000000LL; // Very large number with LL suffix

    std::cout << "Positive count: " << positiveCount << "\n";
    std::cout << "Temperature: " << temperature << "\n";
    std::cout << "Distance: " << distance << "\n";
    std::cout << "Stars in galaxy: " << starsInGalaxy << "\n";

    return 0;
}
```

2.3.7 Summary

- Type modifiers adjust the size and range of integers.
- Use `signed` for values that include negatives; `unsigned` for only non-negative values with a larger positive range.
- Use `short` to save memory with smaller numbers, `long` or `long long` for large values.

- Choosing the right modifier improves portability, memory use, and correctness.

2.4 Type Aliases with `typedef` and `using`

In C++, **type aliases** allow you to create alternative names for existing types. Using aliases can simplify complex type declarations, improve code readability, and make your programs easier to maintain and modify.

2.4.1 Why Use Type Aliases?

- **Improved readability:** Aliases give meaningful names to complicated or frequently used types.
- **Simplified maintenance:** Changing the underlying type only requires updating the alias in one place.
- **Better abstraction:** They help express intent more clearly by naming types according to their roles.

2.4.2 Creating Type Aliases with `typedef`

The traditional way to create a type alias in C++ is using the `typedef` keyword, followed by the existing type and the alias name.

Syntax:

```
typedef existing_type alias_name;
```

Example:

```
typedef unsigned int uint;      // Alias 'uint' for 'unsigned int'
typedef int* IntPtr;            // Alias for pointer to int
typedef void (*FuncPtr)();      // Alias for pointer to function returning void and taking no parameter
```

Usage:

```
uint count = 100;
IntPtr p = nullptr;

void hello() {
    std::cout << "Hello!\n";
}

FuncPtr f = hello;
f();  // Calls hello()
```

2.4.3 Creating Type Aliases with `using` (Modern C)

Since C++11, a new and often preferred way to create type aliases is with the `using` keyword. It is more readable and especially useful for templated or more complex types.

Syntax:

```
using alias_name = existing_type;
```

Example:

```
using uint = unsigned int;        // Alias 'uint' for 'unsigned int'
using IntPtr = int*;              // Alias for pointer to int
using FuncPtr = void(*)();        // Alias for pointer to function returning void and taking no paramete
```

2.4.4 Comparing `typedef` and `using`

Feature	`typedef`	`using`
Syntax	`typedef existing_type alias;`	`using alias = existing_type;`
Readability	Less intuitive for complex types	More readable and clear
Support for templates	Less flexible	Can alias template types
Introduced	Since early C++	Since C++11

2.4.5 Practical Examples

Full runnable code:

```
#include <iostream>
#include <vector>

// Typedef example
typedef std::vector<int> IntVector;

// Using example
using StringVector = std::vector<std::string>;

// Function pointer aliases
typedef void (*FuncPtrOld)();
using FuncPtrNew = void(*)();

void greet() {
    std::cout << "Hello from function pointer!\n";
}

int main() {
    IntVector numbers = {1, 2, 3, 4};
    StringVector names = {"Alice", "Bob", "Carol"};

    FuncPtrOld f1 = greet;
    FuncPtrNew f2 = greet;

    f1();
    f2();

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

    for (const auto& name : names) {
        std::cout << name << " ";
    }
    std::cout << "\n";

    return 0;
}
```

2.4.6 Summary

- Use **`typedef`** or the modern **`using`** keyword to create type aliases.
- Type aliases improve readability, reduce code duplication, and clarify intent.
- Prefer `using` in modern C++ for better readability and template support.
- Aliases are especially helpful for complex types like pointers, function pointers, and template instantiations.

Chapter 3.

Operators and Expressions

1. Arithmetic, Relational, Logical Operators
2. Assignment Operators and Compound Assignments (`+=`, `-=`)
3. Increment and Decrement (`++`, `--`)
4. Bitwise Operators and Shifts
5. Operator Precedence and Associativity
6. Conditional (`?:`) and Comma Operators

3 Operators and Expressions

3.1 Arithmetic, Relational, Logical Operators

Operators are symbols that perform operations on variables and values. In C++, **arithmetic**, **relational**, and **logical** operators are fundamental for performing calculations, comparisons, and decision-making in your programs.

3.1.1 Arithmetic Operators

Arithmetic operators perform basic mathematical calculations. They work on numeric types like `int`, `float`, `double`, and more.

Operator	Description	Example	Result
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`10 - 4`	`6`
`*`	Multiplication	`7 * 2`	`14`
`/`	Division	`20 / 5`	`4`
`%`	Modulo (remainder)	`17 % 5`	`2`

3.1.2 Notes on Arithmetic Operators:

- Division `/` between two integers performs **integer division**, truncating any decimal part.
  ```cpp
  int result = 7 / 3;  // result is 2, fractional part discarded
  ```
- The modulo operator `%` only works with integers and gives the remainder after division.

3.1.3 Examples:

```cpp
int a = 10;
int b = 3;

int sum = a + b;           // 13
int difference = a - b;    // 7
int product = a * b;       // 30
int quotient = a / b;      // 3 (integer division)
int remainder = a % b;     // 1
```

3.1.4 Relational Operators

Relational operators compare two values and return a **boolean** result (`true` or `false`). They are primarily used in conditions such as `if` statements and loops.

Operator	Description	Example	Result
`==`	Equal to	`x == y`	`true` if `x` equals `y`
`!=`	Not equal to	`x != y`	`true` if `x` not equal to `y`
`<`	Less than	`x < y`	`true` if `x` less than `y`
`>`	Greater than	`x > y`	`true` if `x` greater than `y`
`<=`	Less than or equal to	`x <= y`	`true` if `x` `y`
`>=`	Greater than or equal to	`x >= y`	`true` if `x` `y`

3.1.5 Examples:

```cpp
int x = 5;
int y = 10;

bool equal = (x == y);          // false
bool notEqual = (x != y);       // true
bool lessThan = (x < y);        // true
bool greaterThan = (x > y);     // false
bool lessOrEqual = (x <= y);    // true
bool greaterOrEqual = (x >= y);// false
```

These operators are often used in control flow:

```cpp
if (x < y) {
    std::cout << "x is less than y\n";
}
```

3.1.6 Logical Operators

Logical operators combine or modify boolean values. They are essential in forming complex conditions.

Operator	Description	Example	Result
`&&`	Logical AND	`(a > 0) && (b < 5)`	`true` if both conditions true
$\|\|$	Logical OR	$(a == 0)\|\|(b == 0)$	`true` if at least one condition true

Operator	Description	Example	Result
`!`	Logical NOT (negation)	`!(a == b)`	`true` if condition is false

3.1.7 Truth Tables

A	B	A && B	A || B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

3.1.8 Examples:

```cpp
int a = 5;
int b = 10;

bool cond1 = (a < b) && (b < 20);   // true && true => true
bool cond2 = (a == 5) || (b == 5);  // true || false => true
bool cond3 = !(a == b);              // !(false) => true
```

Logical operators are frequently used in `if` statements:

```cpp
if ((a > 0) && (b < 20)) {
    std::cout << "Both conditions are true\n";
}
```

3.1.9 Summary

- **Arithmetic operators** perform basic math operations: `+`, `-`, `*`, `/`, `%`.
- **Relational operators** compare values and return `true` or `false`: `==`, `!=`, `<`, `>`, `<=`, `>=`.
- **Logical operators** combine or negate boolean expressions: `&&` (AND), `||` (OR), and `!` (NOT).
- These operators are fundamental in forming expressions used in calculations and controlling program flow.

3.2 Assignment Operators and Compound Assignments (+=, -=)

In C++, **assignment operators** are used to assign values to variables. The most basic form is the simple assignment operator `=`, but C++ also offers **compound assignment operators** like `+=` and `-=`, which combine an operation with assignment for more concise code.

3.2.1 Simple Assignment (=)

The simple assignment operator assigns the value on the right-hand side to the variable on the left-hand side.

Syntax:

```
variable = expression;
```

Example:

```
int x = 5;      // Assigns 5 to x
x = 10;         // Changes x to 10
```

3.2.2 Compound Assignment Operators

Compound assignment operators perform an operation on a variable and then assign the result back to that same variable. They are shorthand for writing an operation followed by assignment.

Common compound assignment operators:

Operator	Equivalent To	Description
`+=`	`x = x + y`	Add and assign
`-=`	`x = x - y`	Subtract and assign
`*=`	`x = x * y`	Multiply and assign
`/=`	`x = x / y`	Divide and assign
`%=`	`x = x % y`	Modulo and assign (integers)
`&=`	`x = x & y`	Bitwise AND and assign
|=	x = x | y	Bitwise OR and assign
`^=`	`x = x ^ y`	Bitwise XOR and assign
`<<=`	`x = x << y`	Left shift and assign
`>>=`	`x = x >> y`	Right shift and assign

3.2.3 Advantages of Compound Assignments

- **Conciseness:** Reduces code verbosity by combining operation and assignment in a single expression.
- **Readability:** Makes it clearer that a variable is being updated based on its current value.
- **Potential efficiency:** Compilers can optimize compound assignments better than separate operations (though modern compilers often optimize both equivalently).

3.2.4 Examples:

```
int a = 10;
a += 5;      // Equivalent to: a = a + 5;  Result: a = 15

int b = 20;
b -= 7;      // Equivalent to: b = b - 7;  Result: b = 13

int c = 4;
c *= 3;      // Equivalent to: c = c * 3;  Result: c = 12

int d = 15;
d /= 5;      // Equivalent to: d = d / 5;  Result: d = 3

int e = 10;
e %= 3;      // Equivalent to: e = e % 3;  Result: e = 1
```

3.2.5 Using Compound Assignments in Practice

Compound assignments are especially common in loops or algorithms that increment or modify values repeatedly:

```
int sum = 0;
for (int i = 1; i <= 5; ++i) {
    sum += i;    // Adds i to sum on each iteration
}
// sum now equals 15 (1+2+3+4+5)
```

3.2.6 Summary

- The simple assignment operator `=` assigns a value to a variable.
- Compound assignment operators like `+=` and `-=` combine arithmetic operations with assignment for shorter, clearer code.
- They help improve code readability and can enable compiler optimizations.

- Use compound assignments whenever updating a variable based on its current value to keep your code clean and expressive.

3.3 Increment and Decrement (++, --)

The **increment (++)** and **decrement (--)** operators are shorthand operators that increase or decrease an integer variable's value by exactly one. These operators are widely used in loops, counters, and expressions where stepwise changes to values are needed.

3.3.1 Prefix vs Postfix Forms

Both increment and decrement operators come in two forms:

- **Prefix form:** `++x` or `--x`
- **Postfix form:** `x++` or `x--`

The difference lies in **when the variable is updated relative to the expression evaluation**.

3.3.2 How They Work

Operator	Meaning	Effect on `x`	Return Value
`++x`	Prefix increment	Increment `x` by 1	Returns the *new* (incremented) value
`x++`	Postfix increment	Increment `x` by 1	Returns the *original* value of `x`
`--x`	Prefix decrement	Decrement `x` by 1	Returns the *new* (decremented) value
`x--`	Postfix decrement	Decrement `x` by 1	Returns the *original* value of `x`

3.3.3 Examples Demonstrating Prefix and Postfix

```
int x = 5;
int y;

// Prefix increment: increments first, then returns value
y = ++x;  // x becomes 6, y is assigned 6

// Reset x
```

```
x = 5;

// Postfix increment: returns value, then increments
y = x++;  // y is assigned 5, then x becomes 6
```

3.3.4 Detailed Explanation

- **Prefix (++x):** The variable x is increased **before** its value is used in the expression.
- **Postfix (x++):** The original value of x is used in the expression **before** it is incremented.

This distinction becomes important in more complex expressions:

```
int a = 10;
int b = ++a + 5;  // a becomes 11 first, then b = 11 + 5 = 16

int c = 10;
int d = c++ + 5;  // d = 10 + 5 = 15, then c becomes 11
```

3.3.5 Common Use in Loops

Increment and decrement operators are most often used to control loops:

```
// Using prefix increment
for (int i = 0; i < 5; ++i) {
    std::cout << i << " ";
}

// Using postfix increment (functionally equivalent here)
for (int i = 0; i < 5; i++) {
    std::cout << i << " ";
}
```

While both forms work in loops, **prefix increment** is often recommended for iterators and objects because it can be more efficient (though for simple integers, compilers typically generate the same code).

3.3.6 Summary

- ++ and -- increment or decrement a variable by 1.
- **Prefix form (++x)** updates the variable before the value is used.
- **Postfix form (x++)** uses the variable's original value, then updates it.
- Understanding prefix vs postfix is essential when these operators appear in larger expressions.
- Commonly used in loops and counters to succinctly increase or decrease values.

3.4 Bitwise Operators and Shifts

Bitwise operators allow you to manipulate individual bits within integer values. They are powerful tools often used in low-level programming, embedded systems, cryptography, and performance-critical applications where direct control of bits is required.

3.4.1 Bitwise Operators

Operator	Description	Syntax Example	Explanation
`&`	Bitwise AND	`a & b`	Sets each bit to 1 if both bits are 1
|	Bitwise OR	a| b	Sets each bit to 1 if either bit is 1
`^`	Bitwise XOR (exclusive OR)	`a ^ b`	Sets each bit to 1 if bits are different
`~`	Bitwise NOT (one's complement)	`~a`	Inverts all bits

3.4.2 Bit Shifts

Operator	Description	Syntax Example	Explanation
`<<`	Left shift	`a << 2`	Shifts bits of `a` left by 2 positions (multiplies by 4)
`>>`	Right shift	`a >> 3`	Shifts bits of `a` right by 3 positions (divides by 8, floor)

3.4.3 How Bitwise Operators Work

Each bitwise operator performs its operation on the binary representation of integers, bit by bit.

Example:

Consider two 8-bit integers:

```
a = 60;  // binary: 0011 1100
b = 13;  // binary: 0000 1101
```

- **Bitwise AND (`&`)**

```
int result = a & b;  // binary: 0000 1100 (decimal 12)
```

- **Bitwise OR (|)**

```
int result = a | b;  // binary: 0011 1101 (decimal 61)
```

- **Bitwise XOR (^)**

```
int result = a ^ b;  // binary: 0011 0001 (decimal 49)
```

- **Bitwise NOT (~)**

```
int result = ~a;     // binary: 1100 0011 (decimal depends on int size and signedness)
```

3.4.4 Bit Shift Examples

- **Left shift (<<):**

Shifting left by 1 position multiplies by 2.

```
int a = 5;          // binary: 0000 0101
int result = a << 1; // binary: 0000 1010 (decimal 10)
```

- **Right shift (>>):**

Shifting right by 1 divides by 2 (integer division).

```
int a = 20;         // binary: 0001 0100
int result = a >> 2; // binary: 0000 0101 (decimal 5)
```

3.4.5 Typical Applications

- **Bit Masking:** Select or clear specific bits using AND (&) with masks.

```
int flags = 0b10101010;
int mask = 0b00001111;
int masked = flags & mask; // Extract lower 4 bits: 0b00001010 (decimal 10)
```

- **Setting bits:** Use OR (|) to set bits.

```
int flags = 0b00000000;
flags |= 0b00000100; // Set the 3rd bit: flags is now 0b00000100
```

- **Toggling bits:** Use XOR (^) to flip bits.

```
int flags = 0b00000101;
flags ^= 0b00000100; // Toggle 3rd bit: flags becomes 0b00000001
```

- **Clearing bits:** Use AND with negated mask.

```cpp
int flags = 0b00001111;
flags &= ~0b00000100; // Clear the 3rd bit: flags becomes 0b00001011
```

3.4.6 Summary

- Bitwise operators (`&`, `|`, `^`, `~`) manipulate bits individually.
- Bit shifts (`<<`, `>>`) move bits left or right, useful for multiplying or dividing by powers of two.
- These operators are essential for bit masking, setting, clearing, and toggling bits efficiently.
- Understanding bitwise operations unlocks powerful low-level programming techniques.

3.5 Operator Precedence and Associativity

When you write expressions involving multiple operators, C++ uses **operator precedence** and **associativity** rules to determine the order in which operations are performed. Understanding these rules is crucial for writing correct expressions and avoiding subtle bugs.

3.5.1 What Is Operator Precedence?

Operator precedence defines the priority of operators when evaluating an expression. Operators with higher precedence are evaluated **before** operators with lower precedence.

For example, multiplication (`*`) has higher precedence than addition (`+`), so in the expression:

```cpp
int result = 2 + 3 * 4;
```

The multiplication happens first (`3 * 4 = 12`), then addition:

```cpp
result = 2 + 12; // result is 14
```

3.5.2 What Is Associativity?

Associativity defines the order in which operators with the **same precedence** are evaluated. Most operators associate **left-to-right** or **right-to-left**.

- **Left-to-right associativity** means operators are evaluated from left to right.
- **Right-to-left associativity** means operators are evaluated from right to left.

For example, subtraction (`-`) has left-to-right associativity:

```
int x = 10 - 4 - 2;
// Equivalent to (10 - 4) - 2 = 4
```

3.5.3 Common Operator Precedence Table (Simplified)

Precedence Level	Operators	Associativity
1 (highest)	`()` (parentheses)	n/a (grouping)
2	`++`, `--` (prefix), `+`, `-` (unary), `!`, `~`	Right to left
3	`*`, `/`, `%`	Left to right
4	`+`, `-`	Left to right
5	`<<`, `>>` (bit shifts)	Left to right
6	`<`, `<=`, `>`, `>=`	Left to right
7	`==`, `!=`	Left to right
8	`&` (bitwise AND)	Left to right
9	`^` (bitwise XOR)	Left to right
10	| (bitwise OR)	Left to right
11	`&&` (logical AND)	Left to right
12	|| (logical OR)	Left to right
13	`=` and compound assignments (`+=`, `-=` etc.)	Right to left

3.5.4 Using Parentheses to Control Evaluation

Parentheses `()` have the **highest precedence** and can override the default rules, forcing the enclosed expression to be evaluated first.

Example:

```
int result1 = 2 + 3 * 4;        // 2 + (3 * 4) = 14
int result2 = (2 + 3) * 4;      // (2 + 3) * 4 = 20
```

Without parentheses, multiplication happens before addition. Adding parentheses changes the order and result.

3.5.5 More Examples Illustrating Precedence and Associativity

```
int a = 10;
int b = 5;
int c = 2;
int result = a - b - c;  // Evaluated as (a - b) - c = 3

int shifted = a << 2 + 1;
// Evaluated as a << (2 + 1) because '+' has higher precedence than '<<'
// So shifted = 10 << 3 = 80
```

In the last example, without parentheses, `2 + 1` is calculated before the shift. If you wrote:

```
int shifted2 = (a << 2) + 1;  // Evaluated as (10 << 2) + 1 = 40 + 1 = 41
```

3.5.6 Summary

- **Operator precedence** determines which operations are performed first in expressions with multiple operators.
- **Associativity** determines the order of evaluation when operators share the same precedence.
- Use **parentheses** to explicitly control evaluation order and improve code clarity.
- Familiarize yourself with operator precedence to avoid unexpected results and bugs.

3.6 Conditional (`?:`) and Comma Operators

C++ offers some less common but very useful operators that help write concise and expressive code: the **conditional (ternary) operator `?:`** and the **comma operator `,`**. Both have special roles in controlling expression evaluation.

3.6.1 The Conditional (Ternary) Operator `?:`

The conditional operator is a **shorthand if-else** expression that returns one of two values based on a condition.

Syntax

```
condition ? expression_if_true : expression_if_false;
```

- **condition:** An expression that evaluates to `true` or `false`.
- **expression_if_true:** Evaluated and returned if the condition is true.
- **expression_if_false:** Evaluated and returned if the condition is false.

Example

```
int a = 10;
int b = 20;

int max = (a > b) ? a : b;  // max is assigned the larger of a and b
```

Equivalent traditional if-else:

```
int max;
if (a > b) {
    max = a;
} else {
    max = b;
}
```

The conditional operator allows this logic to be written concisely in a single expression.

Usage Tips

- The ternary operator is especially useful in assignments and return statements.
- Avoid overusing nested ternary operators as they can reduce readability.

3.6.2 The Comma Operator ,

The comma operator allows **multiple expressions** to be evaluated in sequence where only one expression is expected. It evaluates each expression from left to right and returns the value of the last expression.

Syntax

```
expression1, expression2, ..., expressionN
```

- All expressions are evaluated in order.
- The **result** of the whole comma expression is the value of `expressionN` (the last one).

Example

```
int a = 1, b = 2, c;

c = (a++, b += 5, a + b);
// Step 1: a++ increments a from 1 to 2 (value discarded)
// Step 2: b += 5 adds 5 to b (now 7)
// Step 3: a + b calculates 2 + 7 = 9
// The value 9 is assigned to c
```

Equivalent longer code:

```
a++;
b += 5;
c = a + b;
```

Practical Uses of Comma Operator

- In `for` loops where multiple expressions need to be executed:

```
for (int i = 0, j = 10; i < j; ++i, --j) {
    std::cout << i << " " << j << "\n";
}
```

- In macros or complex expressions where multiple steps occur in one statement.

3.6.3 Summary

- The **conditional (ternary) operator `?:`** is a concise way to choose between two expressions based on a condition.
- The **comma operator `,`** allows multiple expressions to be evaluated sequentially, returning the last expression's value.
- Both operators enhance expressiveness and can reduce code size, but readability should always be considered.

Chapter 4.
Control Flow

1. `if`, `else if`, `else` Syntax
2. `switch` Statement with `case` and `default`
3. Looping Constructs: `for`, `while`, `do-while`
4. Loop Control: `break`, `continue`, `goto`

4 Control Flow

4.1 `if`, `else if`, `else` Syntax

Conditional branching allows your program to make decisions and execute different code depending on whether certain conditions are true or false. The fundamental tools for this in C++ are the `if`, `else if`, and `else` statements.

4.1.1 Basic Syntax

```cpp
if (condition) {
    // code to execute if condition is true
}
```

- The **condition** is an expression that evaluates to `true` or `false`.
- If the condition is `true`, the code inside the braces `{}` runs.
- If the condition is `false`, the program skips this block.

4.1.2 Adding `else`

```cpp
if (condition) {
    // code if true
} else {
    // code if false
}
```

- If the condition is `false`, the `else` block executes.
- Exactly one of the two blocks runs.

4.1.3 Using `else if` for Multiple Conditions

```cpp
if (condition1) {
    // code if condition1 is true
} else if (condition2) {
    // code if condition1 is false and condition2 is true
} else {
    // code if all above conditions are false
}
```

- `else if` allows you to check multiple exclusive conditions in sequence.
- The first condition that evaluates to `true` runs its block; others are skipped.

4.1.4 Evaluation Logic

- Conditions are evaluated **top to bottom**.
- Once a true condition is found, its block runs, and the rest are ignored.
- If no conditions are true and there is an `else` block, it runs.

4.1.5 Importance of Block Scopes {}

- Blocks `{}` group multiple statements into a single unit.
- Even a single statement following `if` or `else` can be written without braces, but it's recommended to always use braces for clarity and to avoid bugs.

```
if (x > 0)
    std::cout << "Positive\n";  // Valid, but easy to cause errors if more statements are added

if (x > 0) {
    std::cout << "Positive\n";
    // More statements here are clearly part of the block
}
```

4.1.6 Practical Examples

Example 1: Simple Decision

```
int score = 85;

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 {
    std::cout << "Grade: F\n";
}
```

Example 2: Nested Conditions

```
int age = 25;
bool hasID = true;

if (age >= 18) {
    if (hasID) {
        std::cout << "Allowed entry.\n";
    } else {
        std::cout << "ID required.\n";
    }
} else {
    std::cout << "Too young for entry.\n";
}
```

4.1.7 Summary

- Use `if` to test a condition and execute code when it's true.
- Use `else if` to check additional exclusive conditions sequentially.
- Use `else` as a fallback when none of the conditions are met.
- Always use braces `{}` to define blocks clearly and avoid logic errors.
- Nested and sequential conditionals allow complex decision-making in your programs.

4.2 `switch` Statement with `case` and `default`

The `switch` statement is a control flow structure that provides a clean and efficient alternative to multiple `if/else if` conditions when you need to select among many discrete values of an integral expression.

4.2.1 Structure and Syntax

```
switch (expression) {
    case constant1:
        // code executed if expression == constant1
        break;

    case constant2:
        // code executed if expression == constant2
        break;

    // more cases...

    default:
        // code executed if expression does not match any case
}
```

- **`expression`**: Must evaluate to an integral or enumeration type (`int`, `char`, `enum`, etc.).
- **`case constant:`**: Each case label defines a value to compare against `expression`.
- **`break;`**: Terminates the switch and prevents execution from falling through to the next case.
- **`default:`**: Optional. Executes if no matching case is found.

4.2.2 How `switch` Works

1. The `expression` is evaluated once.
2. The program compares the result with each `case` constant.
3. If a match is found, the statements in that case execute.
4. Execution continues until a `break;` is encountered or the end of the switch block.
5. If no case matches, the `default` block (if present) executes.

4.2.3 Importance of `break`

Without `break` statements, C++ executes **fall-through** behavior—code continues into subsequent cases regardless of their conditions. This can lead to unexpected behavior if not carefully controlled.

4.2.4 Example: Using `switch` with `int`

```
int day = 3;

switch (day) {
    case 1:
        std::cout << "Monday\n";
        break;
    case 2:
        std::cout << "Tuesday\n";
        break;
    case 3:
        std::cout << "Wednesday\n";
        break;
    case 4:
        std::cout << "Thursday\n";
        break;
    case 5:
        std::cout << "Friday\n";
        break;
    default:
        std::cout << "Weekend\n";
}
```

Output:

```
Wednesday
```

4.2.5 Example: Demonstrating Fall-Through

```cpp
char grade = 'B';

switch (grade) {
    case 'A':
        std::cout << "Excellent\n";
        break;
    case 'B':
    case 'C':
        std::cout << "Well done\n";
        break;
    case 'D':
        std::cout << "You passed\n";
        break;
    case 'F':
        std::cout << "Better try again\n";
        break;
    default:
        std::cout << "Invalid grade\n";
}
```

Here, cases `'B'` and `'C'` share the same code because there is no `break;` between them, so both trigger `"Well done"`.

4.2.6 Notes and Best Practices

- The `switch` expression must be of an integral or enumeration type. Floating-point types and strings cannot be used directly.
- Always include `break` unless intentional fall-through is desired. To improve readability, comment on intentional fall-through cases.
- The `default` case is optional but recommended to handle unexpected values.
- Cases must use **constant expressions** known at compile-time.

4.2.7 Summary

- `switch` offers a structured way to select one code path among many based on an integral expression.
- Each `case` represents a possible match; `break` prevents falling through to subsequent cases.
- `default` handles unmatched values.
- `switch` is often clearer and more efficient than multiple `if-else` statements when testing one variable against several constant values.

4.3 Looping Constructs: `for`, `while`, `do-while`

Loops are fundamental control structures that allow you to repeatedly execute a block of code while a condition is true or for a fixed number of iterations. C++ provides three primary loop constructs: `for`, `while`, and `do-while`. Each has its own syntax and typical use cases.

4.3.1 The `for` Loop

The `for` loop is used when the number of iterations is known or can be controlled explicitly with an initialization, condition, and update expression.

Syntax:

```
for (initialization; condition; update) {
    // loop body
}
```

- **Initialization:** executed once before the loop starts (e.g., `int i = 0;`).
- **Condition:** checked before each iteration; if false, the loop exits.
- **Update:** executed after each iteration (e.g., increment `i++`).

Example:

```
for (int i = 0; i < 5; ++i) {
    std::cout << "Iteration " << i << "\n";
}
```

Output:

```
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

4.3.2 The `while` Loop

The `while` loop repeats as long as its condition remains true. The condition is evaluated **before** entering the loop body, so the body may not execute at all if the condition is initially false.

Syntax:

```
while (condition) {
    // loop body
}
```

Example:

```cpp
int count = 0;
while (count < 5) {
    std::cout << "Count is " << count << "\n";
    ++count;
}
```

Output:

```
Count is 0
Count is 1
Count is 2
Count is 3
Count is 4
```

4.3.3 The `do-while` Loop

The `do-while` loop guarantees the loop body executes **at least once**, because the condition is evaluated **after** the loop body.

Syntax:

```cpp
do {
    // loop body
} while (condition);
```

Example:

```cpp
int n = 0;
do {
    std::cout << "Number: " << n << "\n";
    ++n;
} while (n < 3);
```

Output:

```
Number: 0
Number: 1
Number: 2
```

4.3.4 Comparison of Loops

Loop Type	Condition Evaluated	Loop Body Executed if Condition is Initially False?	Common Use Case
`for`	Before each iteration	No	Known number of iterations

Loop Type	Condition Evaluated	Loop Body Executed if Condition is Initially False?	Common Use Case
`while`	Before each iteration	No	Condition-controlled loop, possibly zero runs
`do-while`	After each iteration	Yes	Execute at least once, then repeat based on condition

4.3.5 Summary

- Use **`for` loops** when the number of iterations is known or controlled by a counter.
- Use **`while` loops** to repeat code while a condition remains true; the loop may not execute at all.
- Use **`do-while` loops** to ensure the loop body runs at least once before condition checking.
- All loops require careful control of the condition and updates to avoid infinite loops.

4.4 Loop Control: `break`, `continue`, `goto`

Beyond the basic loop structures, C++ provides special statements to control the flow inside loops more precisely: `break`, `continue`, and the less common `goto`. Understanding these helps you write clearer, more efficient loops—but also requires caution to avoid confusing code.

4.4.1 The `break` Statement

- **Purpose:** Immediately exits the closest enclosing loop (`for`, `while`, or `do-while`) or a `switch` statement.
- **Effect:** Stops the loop entirely, transferring control to the statement following the loop.

Example: Exiting a loop early

```cpp
for (int i = 0; i < 10; ++i) {
    if (i == 5) {
        break;  // Exit loop when i is 5
    }
    std::cout << i << " ";
}
```

Output:

```
0 1 2 3 4
```

4.4.2 The `continue` Statement

- **Purpose:** Skips the rest of the current loop iteration and jumps to the start of the next iteration.
- **Effect:** The loop does not exit, but any code after `continue` inside the loop body is ignored for the current iteration.

Example: Skipping an iteration

```
for (int i = 0; i < 5; ++i) {
    if (i == 2) {
        continue;  // Skip printing when i is 2
    }
    std::cout << i << " ";
}
```

Output:

```
0 1 3 4
```

4.4.3 The `goto` Statement

The `goto` statement provides an **unconditional jump** to a labeled statement within the same function.

Syntax:

```
goto label_name;

...

label_name:
    // code
```

Example: Using `goto` to jump

```
int i = 0;

start_loop:
    std::cout << i << " ";
    ++i;
    if (i < 5) {
        goto start_loop;  // Jump back to label, creating a loop
    }
```

Output:

```
0 1 2 3 4
```

4.4.4 Caution with `goto`

- `goto` can make code **hard to read and maintain** because it breaks normal flow control.
- Overuse leads to so-called "spaghetti code" — complex, tangled code that is difficult to follow or debug.
- Modern C++ programming discourages `goto`, favoring structured control flow statements like loops and functions.
- However, `goto` can be useful in some low-level programming or for breaking out of deeply nested loops in rare cases.

4.4.5 Appropriate Use Cases

- **`break`**: Exiting loops early when a condition is met.
- **`continue`**: Skipping unnecessary iterations without exiting the loop.
- **`goto`**: Rare, such as jumping out of nested loops or error handling in some legacy code.

4.4.6 Summary

Keyword	Effect	Typical Use Case
`break`	Exits the nearest enclosing loop or switch	Stop loop early
`continue`	Skips to the next iteration of the loop	Skip part of loop iteration
`goto`	Jumps unconditionally to a labeled statement	Rare, for complex flow or legacy code

4.4.7 Final Example: Combining Loop Control

```
for (int i = 0; i < 10; ++i) {
    if (i == 3) continue;    // Skip 3
    if (i == 7) break;       // Exit at 7
    std::cout << i << " ";
}
```

Output:

```
0 1 2 4 5 6
```

Chapter 5.

Functions and Parameters

1. Function Declaration and Definition Syntax
2. Function Return Types and `void`
3. Parameter Passing: by Value, Reference, and Pointer
4. Default Arguments
5. Inline Functions
6. Overloading Functions

5 Functions and Parameters

5.1 Function Declaration and Definition Syntax

Functions are fundamental building blocks in C++ that allow you to organize, reuse, and modularize code. Understanding how to declare and define functions is essential for writing clear and maintainable programs.

5.1.1 Function Basics

A function in C++ has the following components:

```cpp
return_type function_name(parameter_list);
```

- **`return_type`**: The type of value the function returns (e.g., `int`, `double`, `void`).
- **`function_name`**: The identifier used to call the function.
- **`parameter_list`**: A comma-separated list of parameters (or `void` if none).
- **Function Body**: A block of code enclosed in `{}` where the function's behavior is defined.

5.1.2 Declaration vs. Definition

C++ allows you to **declare** a function separately from its **definition**. This is important for organizing code, especially across multiple files.

Function Declaration (Prototype)

A **declaration** tells the compiler the function's name, return type, and parameters—**but not** what it does.

```cpp
int add(int a, int b);  // Declaration
```

- Ends with a semicolon ;
- Usually placed at the top of a file or in a header file

Function Definition

A **definition** provides the full implementation of the function.

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

- Contains the actual body of the function
- Can appear after the `main()` function, in a separate file, or even inline

5.1.3 Example: Declaration and Definition

Full runnable code:

```
#include <iostream>

// Declaration (prototype)
void greet();

int main() {
    greet();  // Call the function
    return 0;
}

// Definition
void greet() {
    std::cout << "Hello from a function!\n";
}
```

Output:

```
Hello from a function!
```

5.1.4 Function with Parameters and Return Value

```
// Declaration
double multiply(double x, double y);

// Definition
double multiply(double x, double y) {
    return x * y;
}
```

5.1.5 Best Practices

- Place function **declarations** in header files (`.h`) when functions are used in multiple source files.
- Match parameter types and order exactly between declarations and definitions.
- Always provide a return value if the return type is not `void`.

5.1.6 Summary

- A **function declaration** introduces a function's name, return type, and parameters to the compiler.

- A **function definition** provides the complete implementation.
- Use declarations to organize code, especially in multi-file programs.
- Keep syntax consistent to avoid compiler errors.

5.2 Function Return Types and `void`

In C++, every function has a **return type** that specifies what kind of value the function will produce when called. Understanding how return types work is essential for designing functions that correctly communicate data and perform useful tasks.

5.2.1 What Is a Return Type?

The return type is written **before** the function name in its declaration or definition:

```
int add(int a, int b);    // returns an int
void greet();             // returns nothing
double square(double x);  // returns a double
```

- The return type tells the compiler what **type of value** the function will return to the caller.
- If a function doesn't return anything, use the keyword `void`.

5.2.2 Returning Values from Functions

To return a value, use the `return` keyword followed by an expression of the correct type.

Example: Returning an `int`

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

Usage:

```
int result = add(3, 4);  // result becomes 7
```

Example: Returning a `double`

```
double square(double x) {
    return x * x;
}
```

Usage:

```cpp
std::cout << square(2.5);  // Output: 6.25
```

5.2.3 The `void` Return Type

When a function doesn't return a value, it must be declared with the `void` return type. Such functions are typically used for actions like printing output, updating data, or performing side effects.

Example: `void` Function

```cpp
void greet() {
    std::cout << "Hello!\n";
}
```

Usage:

```cpp
greet();  // Simply calls the function; no return value
```

You **cannot** assign the result of a `void` function to a variable:

```cpp
int x = greet();  // NO Error: greet() returns void
```

5.2.4 Returning Other Data Types

Functions can return other types like `char`, `bool`, or even user-defined types such as `structs` and `classes`.

Example: Returning `bool`

```cpp
bool isEven(int n) {
    return n % 2 == 0;
}
```

Usage:

```cpp
if (isEven(4)) {
    std::cout << "Even number.\n";
}
```

5.2.5 Summary

- Every function has a **return type**, specified before the function name.
- The `return` statement sends a value back to the caller.

- Use `void` when a function **does not return anything**.
- Use meaningful return types to communicate results and status.
- Return types must match the type of value returned in the function body.

5.3 Parameter Passing: by Value, Reference, and Pointer

When calling a function in C++, **arguments** (values or variables) are passed to **parameters** (function inputs) using one of three primary methods:

1. **By value** – A copy of the argument is made.
2. **By reference** – The actual variable is passed via an alias.
3. **By pointer** – The memory address of the variable is passed.

Understanding these methods is crucial for controlling how data is shared or modified between functions.

5.3.1 Passing by Value

When a parameter is passed **by value**, a **copy** of the argument is made. Changes to the parameter inside the function do **not** affect the original variable.

Syntax

```
void increment(int x) {
    x = x + 1;
}
```

Usage:

```
int a = 5;
increment(a);
std::cout << a;  // Output: 5 (unchanged)
```

YES **Use when:** You do not want to modify the original value.

5.3.2 Passing by Reference (`&`)

When a parameter is passed **by reference**, the function receives a reference (alias) to the original variable. Modifications affect the **original** argument.

Syntax

```
void increment(int& x) {
    x = x + 1;
}
```

Usage:

```
int a = 5;
increment(a);
std::cout << a;  // Output: 6 (changed)
```

YES **Use when:** You want to modify the caller's variable or avoid copying large data.

5.3.3 Passing by Pointer (*)

When a parameter is passed **by pointer**, the function receives the **memory address** of the variable. The pointer must be dereferenced (*) to access or modify the value.

Syntax

```
void increment(int* x) {
    *x = *x + 1;
}
```

Usage:

```
int a = 5;
increment(&a);  // Pass the address of a
std::cout << a;  // Output: 6 (changed)
```

YES **Use when:** You want to modify the variable and handle `nullptr`, arrays, or dynamic memory.

5.3.4 Comparison Table

Method	Function Syntax	Call Syntax	Can Modify Original?	Notes
By Value	`void f(int x)`	`f(a);`	NO No	Copy made; safe for read-only usage
By Reference	`void f(int& x)`	`f(a);`	YES Yes	Simple, safe; avoids pointer syntax
By Pointer	`void f(int* x)`	`f(&a);`	YES Yes	Explicit address passing; needs `*` and `&`

5.3.5 Example: Modifying Multiple Values

```cpp
void swap(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
```

Usage:

```cpp
int a = 3, b = 7;
swap(a, b);
std::cout << a << ", " << b;  // Output: 7, 3
```

5.3.6 Best Practices

- Prefer **pass-by-value** for small types like `int`, `char`, `bool`.
- Use **pass-by-reference** when modifying variables or avoiding copies.
- Use **pointers** when working with dynamic memory, optional values, or `nullptr`.

5.3.7 Summary

- **By value**: Copies the argument; safe and non-modifying.
- **By reference**: Passes a reference; efficient and allows modification.
- **By pointer**: Passes an address; flexible but requires careful syntax.

Choosing the right method improves safety, clarity, and performance in your functions.

5.4 Default Arguments

Default arguments in C++ allow function parameters to have **predefined values**. When a caller omits those arguments, the function automatically substitutes the defaults. This feature simplifies function calls and supports backward compatibility as functions evolve.

5.4.1 Syntax of Default Arguments

A default argument is assigned in the function **declaration** or **definition** using the `=` operator:

```
void greet(std::string name = "Guest");
```

If no argument is provided for `name`, the function will use `"Guest"` as the default value.

5.4.2 Example: Function with a Default Parameter

Full runnable code:

```
#include <iostream>

void greet(std::string name = "Guest") {
    std::cout << "Hello, " << name << "!\n";
}

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

5.4.3 Multiple Default Arguments

You can define **multiple** default arguments, starting from the **rightmost** parameter. C++ does **not** allow default arguments to precede non-defaulted ones.

Valid:

```
void logMessage(std::string message, int level = 1);
```

Invalid (will not compile):

```
void logMessage(int level = 1, std::string message);  // NO Error
```

5.4.4 Declaration vs. Definition

Default values are usually given **only in the declaration**, especially in header files. Repeating them in both the declaration and definition can cause ambiguity or duplication.

Header (declaration with default):

```
void display(std::string text, int width = 80);
```

Source file (definition without default):

```cpp
void display(std::string text, int width) {
    std::cout << "[" << text << "] width=" << width << "\n";
}
```

5.4.5 Practical Use Case: Backward Compatibility

Imagine a function originally written as:

```cpp
void drawRectangle(int width, int height);
```

Later, you want to support an optional color:

```cpp
void drawRectangle(int width, int height, std::string color = "black");
```

This allows existing code that only passes two arguments to continue working without modification.

5.4.6 Summary

- **Default arguments** reduce redundancy in function calls and make APIs easier to use.
- Must be defined from **right to left** in the parameter list.
- Typically declared in **function prototypes** (header files) only.
- Useful for evolving functions while maintaining **backward compatibility**.

5.5 Inline Functions

The `inline` keyword in C++ is a request to the compiler to **substitute the function's code directly at the point of the call**, instead of generating a normal function call. This can improve performance by eliminating the overhead of a call, especially for **very small functions**.

5.5.1 Purpose of `inline`

Normally, calling a function involves:

1. Saving the current execution state.
2. Jumping to the function's code.

3. Returning back after execution.

For small functions, this overhead may be **more expensive** than executing the actual function body. Inlining can eliminate this overhead by replacing the function call with the **actual function code** during compilation.

5.5.2 Syntax

Use the `inline` keyword before the function definition:

```cpp
inline int square(int x) {
    return x * x;
}
```

Alternatively, when defining a function inside a class, it is **implicitly** considered inline:

```cpp
class Math {
public:
    int cube(int x) { return x * x * x; }  // Inline by default
};
```

5.5.3 Example: Inline Function

Full runnable code:

```cpp
#include <iostream>

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

int main() {
    std::cout << add(2, 3);  // Output: 5
}
```

If the compiler inlines `add(2, 3)`, the code becomes:

```cpp
std::cout << (2 + 3);
```

5.5.4 When to Use `inline`

YES **Good use cases:**

- Very small functions (1–3 lines).
- Frequently called utility functions.

- Accessor methods in classes (`getX()`, `setValue()`).

Avoid inlining when:

- The function is large (can bloat binary code).
- The function is recursive.
- The function is unlikely to be called frequently.

Note: The `inline` keyword is only a **suggestion** to the compiler. Modern compilers decide automatically whether to inline a function or not based on optimization settings, regardless of the keyword.

5.5.5 Benefits and Trade-Offs

Benefit	Trade-Off
Avoids function call overhead	Can increase binary size (code bloat)
Faster execution for tiny code	May reduce cache efficiency

5.5.6 Summary

- Use `inline` to suggest substituting the function body at the call site.
- Best for small, frequently used functions where performance matters.
- Avoid inlining large or complex functions.
- Compilers may ignore `inline` if they decide inlining is inappropriate.

5.6 Overloading Functions

Function overloading allows multiple functions to share the same name **as long as their parameter lists differ** in number or types. This is a form of **polymorphism**, enabling more expressive and intuitive function usage in C++.

5.6.1 Purpose of Overloading

Function overloading improves code **readability** and **consistency** by allowing related operations to use the same function name.

For example:

```
int max(int a, int b);
double max(double a, double b);
```

Both functions are logically performing the same operation—returning the maximum—but for different types.

5.6.2 Rules for Function Overloading

To overload functions, the **signature** must be different. The **signature** includes:

- The number of parameters
- The types of parameters
- The order of parameters

The return type **alone** is **not sufficient** for overloading.

Valid Overloads:

```
void print(int x);
void print(double x);
void print(const std::string& s);
void print(int x, int y);
```

Invalid Overload:

```
int add(int x, int y);
double add(int x, int y);  // NO Error: same signature, different return type
```

5.6.3 Example: Overloaded `print()` Function

Full runnable code:

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

void print(int x) {
    std::cout << "Integer: " << x << "\n";
}

void print(double d) {
    std::cout << "Double: " << d << "\n";
}

void print(const std::string& s) {
    std::cout << "String: " << s << "\n";
}

int main() {
    print(42);              // Calls print(int)
    print(3.14);            // Calls print(double)
    print("Hello");         // Calls print(const std::string&)
}
```

Output:

```
Integer: 42
Double: 3.14
String: Hello
```

5.6.4 Overload Resolution

When you call an overloaded function, the compiler selects the **best match** based on:

- **Exact match** (preferred)
- **Standard type conversions** (e.g., `int` to `double`)
- **User-defined conversions** (lowest priority)

If no suitable match exists or if ambiguity arises, the compiler will emit an error.

Example of Ambiguity:

```
void f(int);
void f(double);

f('A');  // char can convert to both int and double → ambiguous if not resolved
```

5.6.5 Summary

- Function **overloading** allows multiple functions with the **same name** but **different parameter lists**.
- The return type alone **cannot** distinguish overloaded functions.
- Compiler uses **overload resolution** to pick the best match.
- Avoid ambiguous overloads that rely on implicit type conversions.

Function overloading is a powerful feature that promotes clean and readable code. Use it to unify logically related functions while maintaining type safety and clarity.

Chapter 6.

Arrays and Strings

6 Arrays and Strings

6.1 Array Declaration and Initialization

Arrays in C++ are **fixed-size collections** of elements of the same type, stored in **contiguous memory locations**. They are useful for working with groups of related values, such as a list of numbers or characters.

6.1.1 Declaring Arrays

The basic syntax to declare an array is:

```
type arrayName[arraySize];
```

- `type`: The data type of the array elements (e.g., `int`, `char`, `double`)
- `arrayName`: The name of the array
- `arraySize`: A constant expression specifying the number of elements

Example:

```cpp
int numbers[5];  // Declares an array of 5 integers
```

This creates space for five integers: `numbers[0]` through `numbers[4]`.

6.1.2 Initializing Arrays

You can initialize arrays at the time of declaration.

Full Initialization:

```cpp
int values[3] = {10, 20, 30};
```

Each element is explicitly initialized to a value.

Partial Initialization:

```cpp
int values[5] = {1, 2};  // Remaining elements initialized to 0
```

Equivalent to:

```
values[0] = 1
values[1] = 2
values[2] = 0
values[3] = 0
```

```
values[4] = 0
```

Implicit Size Deduction:

If you provide an initializer list, you can omit the size:

```cpp
int values[] = {4, 5, 6};  // Compiler infers size as 3
```

6.1.3 Default Values

- Arrays of **fundamental types** (like `int`, `float`, `char`) declared **without initialization** contain **garbage (undefined)** values.
- You can zero-initialize an array like this:

```cpp
int zeros[5] = {};  // All elements set to 0
```

6.1.4 Accessing Array Elements

Array elements are accessed using an index starting at 0:

```cpp
std::cout << values[0];  // First element
values[2] = 100;         // Assign a value to the third element
```

6.1.5 Arrays in Memory

- Arrays are stored in **contiguous memory**.
- Accessing elements by index is **constant-time**.
- Memory layout for `int arr[4] = {1, 2, 3, 4};` looks like:

```
| 1 | 2 | 3 | 4 |
 ^   ^   ^   ^

|   |   |   |
0   1   2   3   <- Indices
```

The base address `&arr[0]` points to the first element.

6.1.6 Limitations of Fixed-Size Arrays

- **Size must be known at compile time** for static arrays.
- **Cannot resize** after declaration.

- **No bounds checking**—accessing out-of-range indices causes **undefined behavior**.

```
int arr[3] = {1, 2, 3};
std::cout << arr[10];  // NO Undefined behavior!
```

For more dynamic needs, C++ provides alternatives like `std::vector`.

6.1.7 Summary

- Arrays group elements of the same type with a fixed size.
- Elements are stored contiguously in memory and indexed from 0.
- Initialization can be full, partial, or defaulted to zero.
- Static arrays are limited in flexibility and require careful index management.

6.2 Multi-dimensional Arrays

C++ supports **multi-dimensional arrays**, which are arrays of arrays. These are especially useful for representing data in grid-like structures such as **matrices**, **tables**, or **2D game boards**.

6.2.1 Declaring Multi-dimensional Arrays

The general syntax for a two-dimensional array is:

```
type arrayName[rows][columns];
```

Example:

```
int matrix[3][4];  // 3 rows and 4 columns
```

This creates a grid with 12 `int` elements arranged as 3 rows × 4 columns.

6.2.2 Initializing Multi-dimensional Arrays

You can initialize a multi-dimensional array at declaration time using nested braces:

Fully Specified Initialization:

```
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

This layout corresponds to:

```
Row 0: 1 2 3
Row 1: 4 5 6
```

Partially Initialized:

```
int matrix[2][3] = {
    {1},     // Remaining values are 0
    {4, 5}
};
```

Equivalent to:

```
1 0 0
4 5 0
```

Implicit Row Size Deduction:

You can omit the outer size (number of rows) if initializing with nested arrays:

```
int matrix[][3] = {
    {10, 20, 30},
    {40, 50, 60}
};  // Compiler infers 2 rows
```

> YES **Note:** Only the first dimension (rows) can be omitted; column sizes must be specified.

6.2.3 Accessing Elements

To access an element, use a double index:

```
matrix[0][1] = 99;          // Sets value at first row, second column
std::cout << matrix[1][2];  // Accesses second row, third column
```

6.2.4 Example: 2D Array as a Table

Full runnable code:

```
#include <iostream>

int main() {
    int table[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    for (int row = 0; row < 2; ++row) {
        for (int col = 0; col < 3; ++col) {
            std::cout << table[row][col] << " ";
        }
        std::cout << "\n";
    }
}
```

Output:

```
1 2 3
4 5 6
```

6.2.5 Higher-dimensional Arrays

C++ supports arrays with more than two dimensions:

```
int cube[2][3][4];  // 3D array: 2 × 3 × 4
```

Access is done with three indices:

```
cube[1][2][3] = 10;
```

While technically supported, higher-dimensional arrays become harder to manage and are less commonly used than 2D arrays.

6.2.6 Memory Layout

Multi-dimensional arrays in C++ are stored in **row-major order**: rows are stored sequentially in memory.

Given:

```
int grid[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Memory layout is:

```
[1][2][3][4][5][6]
```

6.2.7 Summary

- Multi-dimensional arrays are declared with multiple index brackets (`[][]`).
- Initialization uses nested braces for each row (or layer).
- Access elements using multiple indices (`array[row][col]`).
- Use for tabular data like matrices and grids.
- Data is stored in **row-major** order (leftmost index changes slowest).

6.3 C-style Strings (`char` arrays)

Before the introduction of the `std::string` class, strings in C and early C++ were handled using **C-style strings**—arrays of characters terminated by a special **null character** (`'\0'`). These strings are still widely used in low-level code and C++ libraries that interface with C.

6.3.1 C-style String Representation

A C-style string is a **null-terminated character array**:

```
char greeting[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- The null terminator (`'\0'`) marks the **end of the string**.
- Without the terminator, functions that operate on strings may read **past the array's bounds**, leading to undefined behavior.

6.3.2 String Literal Initialization

C++ provides a convenient shorthand for initializing C-style strings using **string literals**:

```
char message[] = "Hello";
```

This is equivalent to:

```
char message[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- The compiler automatically appends the null terminator.
- You can also use fixed-size arrays:

```
char fixed[10] = "Hi";
```

This results in:

```
'H' 'i' '\0' '\0' '\0' '\0' '\0' '\0' '\0' '\0'
```

6.3.3 Common Pitfalls

YES **Always** ensure space for the null terminator when declaring arrays:

```
char name[5] = "John";  // OK: 4 characters + 1 for '\0'
```

```
char name[4] = "John";  // NO Error: no room for null terminator
```

YES Be cautious when copying or manipulating strings manually—off-by-one errors are easy to make.

6.3.4 Standard Library Functions for C-style Strings

The C++ Standard Library includes `<cstring>` (from C's `<string.h>`) with useful functions for working with `char` arrays:

Function	Description
`strlen(s)`	Returns the length (excluding `'\0'`)
`strcpy(dst, src)`	Copies string from `src` to `dst`
`strcmp(a, b)`	Compares two strings (returns 0 if equal)
`strcat(dst, src)`	Appends `src` to the end of `dst`

Example: Using `strlen` and `strcpy`

Full runnable code:

```
#include <iostream>
#include <cstring>

int main() {
    char source[] = "Hello";
    char destination[10];

    std::strcpy(destination, source);  // Copy string
    std::cout << "Copied: " << destination << "\n";
    std::cout << "Length: " << std::strlen(destination) << "\n";
}
```

6.3.5 Character-by-character Access

Just like arrays, you can access characters using indices:

```
char word[] = "Code";
std::cout << word[0];  // Output: C
word[1] = 'A';         // Changes 'o' to 'A'
```

6.3.6 Example: Input and Output

Full runnable code:

```
#include <iostream>

int main() {
    char name[20];
    std::cout << "Enter your name: ";
    std::cin >> name;
    std::cout << "Hello, " << name << "!\n";
}
```

> WARNING Note: `std::cin >> name;` reads input **until the first whitespace**. To read full lines, use `std::cin.getline(name, size);`.

6.3.7 Summary

- C-style strings are arrays of `char` ending with a null terminator (`'\0'`).
- Use string literals for easy initialization: `"Hello"` is automatically null-terminated.
- String manipulation requires care to avoid memory and boundary issues.
- `<cstring>` provides basic functions like `strcpy`, `strlen`, and `strcmp`.

6.4 `std::string` Basic Syntax and Operations

C++ provides the `std::string` class in the standard library as a safer, more convenient, and more powerful alternative to C-style character arrays. It manages memory automatically, prevents many common string-related bugs, and offers a rich set of member functions for string manipulation.

To use `std::string`, include the `<string>` header and use the `std` namespace.

6.4.1 Declaring and Initializing Strings

```
#include <string>

std::string name = "Alice";
std::string empty;              // Empty string
std::string copy(name);         // Copy initialization
std::string repeat(5, 'x');     // "xxxxx"
```

6.4.2 Assigning Strings

You can assign a string literal or another `std::string` value:

```
std::string greeting;
greeting = "Hello";
greeting = name;  // name is another std::string
```

6.4.3 Concatenation

You can concatenate strings using the `+` operator or `+=`:

```
std::string first = "Hello";
std::string second = "World";
std::string combined = first + " " + second;  // "Hello World"
first += " there";                             // first becomes "Hello there"
```

6.4.4 Accessing Characters

Like arrays, `std::string` supports indexing:

```
std::string word = "Code";
char c = word[0];       // 'C'
word[1] = 'A';          // word becomes "CAde"
```

Use `.at(index)` for bounds-checked access:

```
char ch = word.at(2);  // 'd'
// word.at(10);        // Throws std::out_of_range
```

6.4.5 Common Member Functions

Function	Description
`length()` or `size()`	Returns number of characters
`empty()`	Returns true if the string is empty
`clear()`	Erases all characters from the string
`substr(pos, len)`	Returns a substring
`find(str)`	Returns position of `str` or `npos` if not found
`replace(pos, len, s)`	Replaces part of the string
`append(str)`	Adds `str` to the end

6.4.6 Example: Basic String Operations

Full runnable code:

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

int main() {
    std::string name = "Alice";
    std::string greeting = "Hello, " + name;

    std::cout << greeting << "\n";           // Hello, Alice
    std::cout << "Length: " << greeting.size() << "\n";

    greeting.replace(0, 5, "Hi");            // Replace "Hello" with "Hi"
    std::cout << greeting << "\n";           // Hi, Alice

    std::string part = greeting.substr(4);   // Extract "Alice"
    std::cout << "Name part: " << part << "\n";
}
```

6.4.7 String Input and Output

`std::cin` stops at the first whitespace. Use `std::getline` to read full lines:

Full runnable code:

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

int main() {
    std::string input;
    std::cout << "Enter a line: ";
    std::getline(std::cin, input);
    std::cout << "You entered: " << input << "\n";
}
```

6.4.8 Comparison

Strings can be compared using relational operators:

```
std::string a = "apple";
std::string b = "banana";

if (a < b) {
    std::cout << a << " comes before " << b << "\n";
}
```

6.4.9 Summary

- `std::string` is a standard library class for representing strings safely and flexibly.
- Supports direct assignment, concatenation, and element access.
- Provides powerful built-in functions for searching, modifying, and comparing strings.
- Automatically manages memory, unlike C-style `char` arrays.

Chapter 7.

Pointers and References

1. Pointer Syntax and Declaration
2. Pointer Dereferencing (`*`) and Address-of (`&`) Operators
3. Pointer Arithmetic
4. Null Pointers and `nullptr`
5. References and Reference Variables (`&`)
6. Pointer to Pointer and Reference to Reference

7 Pointers and References

7.1 Pointer Syntax and Declaration

Pointers are variables that **store the memory address** of another variable. They provide powerful capabilities such as dynamic memory management, efficient array handling, and function parameter passing by reference.

7.1.1 Declaring Pointers

To declare a pointer, specify the type of the variable it will point to, followed by an asterisk (*), then the pointer's name:

```
type* pointerName;
```

- `type` is the type of the object the pointer points to.
- The asterisk * indicates this variable is a pointer.
- The pointer stores the **address** of a variable of the specified type.

Examples:

```
int* p;       // Pointer to an int
double* dp;   // Pointer to a double
char* cp;     // Pointer to a char
```

WARNING The * binds to the variable name, so `int* p, q;` declares `p` as a pointer to int but `q` as a plain int.

7.1.2 Initializing Pointers

Pointers should be initialized with the **address** of a variable using the **address-of operator** (&):

```
int x = 42;
int* p = &x;  // p stores the address of x
```

The pointer `p` now holds the memory address where `x` resides.

7.1.3 Difference Between Pointers and Regular Variables

- **Regular variable**: Holds a **value** directly.
- **Pointer variable**: Holds a **memory address** pointing to a value.

Example:

```
int x = 10;    // x holds the value 10
int* ptr = &x; // ptr holds the address of x
```

Printing these:

```
std::cout << "x = " << x << "\n";          // Outputs: 10
std::cout << "ptr = " << ptr << "\n";      // Outputs: some memory address
std::cout << "*ptr = " << *ptr << "\n";    // Dereferences ptr to get value: 10
```

7.1.4 Example: Pointer Declaration and Assignment

Full runnable code:

```
#include <iostream>

int main() {
    int num = 100;
    int* pNum = &num;   // Declare pointer and initialize with address of num

    std::cout << "Value of num: " << num << "\n";
    std::cout << "Address of num: " << &num << "\n";
    std::cout << "Value of pNum (address stored): " << pNum << "\n";
    std::cout << "Dereferenced pNum (*pNum): " << *pNum << "\n";

    return 0;
}
```

7.1.5 Summary

- Pointers hold **addresses** of variables, not values themselves.
- Use `type* ptr` syntax to declare a pointer to `type`.
- Initialize pointers with the address of a variable using `&`.
- Pointers enable indirect access and manipulation of variables via their memory addresses.

7.2 Pointer Dereferencing (*) and Address-of (&) Operators

Two fundamental operators related to pointers are the **address-of operator (`&`)** and the **dereference operator (`*`)**. Understanding their usage is essential for working effectively with pointers.

7.2.1 Address-of Operator (&)

The address-of operator returns the **memory address** of a variable.

Syntax:

```
&variable
```

Example:

```
int x = 10;
int* p = &x;   // p stores the address of x
```

Here, `&x` is the address of variable `x`, which is assigned to pointer `p`.

7.2.2 Dereference Operator (*)

The dereference operator accesses the **value stored at the memory address** held by a pointer.

- Applying `*` to a pointer variable accesses or modifies the variable it points to.

Syntax:

```
*pointer
```

Example:

```
int x = 10;
int* p = &x;

std::cout << *p;  // Outputs: 10

*p = 20;          // Changes the value of x through the pointer
std::cout << x;   // Outputs: 20
```

7.2.3 Reading and Writing Through Pointers

Using the dereference operator, you can **read** or **modify** the value of the pointed-to variable.

```
int value = 5;
int* ptr = &value;

// Reading
int readValue = *ptr;  // readValue is now 5

// Writing
```

```cpp
*ptr = 15;             // value is now changed to 15
```

7.2.4 Complete Example

Full runnable code:

```cpp
#include <iostream>

int main() {
    int number = 42;
    int* pNumber = &number;  // Pointer holds address of number

    std::cout << "number = " << number << "\n";       // 42
    std::cout << "pNumber (address) = " << pNumber << "\n";
    std::cout << "*pNumber (value) = " << *pNumber << "\n";  // Dereference to get value

    // Modify value through pointer
    *pNumber = 100;
    std::cout << "number after *pNumber = 100: " << number << "\n";  // 100

    return 0;
}
```

7.2.5 Summary

Operator	Purpose	Example	Result
`&`	Returns the **address** of a variable	`int* p = &x;`	`p` stores address of `x`
`*`	Accesses the **value** at a pointer's address	`int val = *p;`	`val` gets the value pointed by `p`

Understanding these operators enables indirect access to variables and is the foundation for dynamic data structures, passing arguments by reference, and more advanced C++ programming techniques.

7.3 Pointer Arithmetic

Pointer arithmetic allows you to perform operations on pointers to navigate through contiguous blocks of memory, such as arrays. Understanding how arithmetic works on pointers is essential for efficient low-level data access and manipulation.

7.3.1 Basics of Pointer Arithmetic

Unlike regular integers, pointer arithmetic takes into account the **size of the data type** the pointer points to. When you add or subtract an integer to/from a pointer, the pointer moves by that many elements, **not bytes**.

7.3.2 Common Pointer Arithmetic Operations

Operation	Description
`ptr++`	Moves pointer to the **next** element
`ptr--`	Moves pointer to the **previous** element
`ptr + n` or `ptr - n`	Moves pointer forward or backward by `n` elements
`ptr1 - ptr2`	Returns the number of elements between two pointers

7.3.3 How Size of Pointed Type Affects Arithmetic

For example, if `ptr` points to an `int` and the size of `int` is 4 bytes:

- `ptr + 1` advances the pointer by `4` bytes (one `int`).
- `ptr + 2` advances by `8` bytes (two `int`s).

If `ptr` points to a `double` (usually 8 bytes), `ptr + 1` advances by `8` bytes.

7.3.4 Example: Traversing an Array with a Pointer

Full runnable code:

```cpp
#include <iostream>

int main() {
    int numbers[] = {10, 20, 30, 40, 50};
    int* ptr = numbers;  // points to the first element

    for (int i = 0; i < 5; ++i) {
        std::cout << "Element " << i << " = " << *ptr << "\n";
        ptr++;  // move to the next integer in the array
    }
}
```

Output:

```
Element 0 = 10
Element 1 = 20
Element 2 = 30
Element 3 = 40
Element 4 = 50
```

7.3.5 Pointer Subtraction Example

You can also subtract one pointer from another to find the number of elements between them:

```
int arr[5] = {1, 2, 3, 4, 5};
int* start = &arr[0];
int* end = &arr[4];

std::cout << "Distance between pointers: " << (end - start) << "\n";  // Outputs 4
```

7.3.6 Important Notes

- Pointer arithmetic is **only valid within arrays or contiguous memory blocks**.
- Adding or subtracting pointers pointing to unrelated memory leads to undefined behavior.
- You cannot perform arithmetic directly on `void*` pointers because the size of the pointed type is unknown.

7.3.7 Summary

- Pointer arithmetic advances or retreats a pointer by a number of elements (not bytes).
- Increment (`ptr++`) and decrement (`ptr--`) move the pointer to the next or previous element.
- Adding/subtracting integers moves the pointer by that many elements.
- Subtracting pointers returns the number of elements between them.

7.4 Null Pointers and `nullptr`

In C++, a **null pointer** is a special pointer value that indicates the pointer **does not point to any valid object or memory location**. Null pointers are essential for safe pointer initialization and error handling.

7.4.1 What is a Null Pointer?

A null pointer acts as a sentinel value to represent "no address" or "empty" pointer. It prevents pointers from unintentionally pointing to arbitrary or invalid memory, which can lead to undefined behavior and program crashes.

7.4.2 Traditional Null Pointer: `NULL`

Historically, C and C++ used the macro `NULL` (usually defined as `0` or `((void*)0)`) to represent null pointers:

```
int* p = NULL;  // p points to nothing
```

However, `NULL` is essentially an integer constant zero, which can sometimes cause ambiguity, especially in function overload resolution.

7.4.3 Modern Null Pointer: `nullptr` (Since C11)

C++11 introduced the **`nullptr`** keyword as a **dedicated null pointer constant**:

```
int* p = nullptr;  // Preferred way to initialize null pointers in modern C++
```

`nullptr` is strongly typed and resolves ambiguity better than `NULL`. It should be preferred in all new code.

7.4.4 Safe Pointer Initialization

Always initialize pointers to `nullptr` (or `NULL` in legacy code) if you don't yet have a valid address to assign:

```
int* p = nullptr;  // Safe initialization
```

This practice avoids undefined behavior caused by **dangling** or **wild pointers** (uninitialized pointers pointing to unknown memory).

7.4.5 Checking for Null Pointers

Before dereferencing a pointer, it's good practice to check if it is not null:

```
if (p != nullptr) {
    // Safe to dereference
    std::cout << *p << "\n";
} else {
    std::cout << "Pointer is null, cannot dereference.\n";
}
```

You can also write this condition more concisely:

```
if (p) {
    // pointer is not null
}
```

7.4.6 Example: Using `nullptr` and Null Checks

Full runnable code:

```
#include <iostream>

void printValue(int* ptr) {
    if (ptr != nullptr) {
        std::cout << "Value pointed to: " << *ptr << "\n";
    } else {
        std::cout << "Null pointer received.\n";
    }
}

int main() {
    int x = 42;
    int* validPtr = &x;
    int* nullPtr = nullptr;

    printValue(validPtr);  // Output: Value pointed to: 42
    printValue(nullPtr);   // Output: Null pointer received.

    return 0;
}
```

7.4.7 Summary

- Null pointers indicate that a pointer does **not** point to any valid memory.
- Use `nullptr` (introduced in C++11) for a type-safe null pointer constant.
- Initialize pointers to `nullptr` to avoid uninitialized or dangling pointers.
- Always check pointers against `nullptr` before dereferencing to prevent runtime errors.

7.5 References and Reference Variables (`&`)

References provide an alternative way to access variables indirectly, similar to pointers, but with a simpler and safer syntax. They are widely used in C++ for parameter passing, return values, and aliasing variables.

7.5.1 What Is a Reference?

A **reference** is an alias for an existing variable. Once initialized to refer to a variable, a reference acts as another name for that variable.

- Unlike pointers, references **cannot be null** or reassigned to refer to a different object.
- They provide **simpler syntax** because you use them just like normal variables, without needing to dereference.

7.5.2 Declaring Reference Variables

You declare a reference by placing an ampersand (`&`) after the type:

```
type& referenceName = variable;
```

- The reference must be **initialized immediately** upon declaration.
- After initialization, it cannot be changed to refer to another variable.

Example:

```
int x = 10;
int& ref = x;   // ref is a reference (alias) to x
```

Any use of `ref` is effectively the same as using `x`.

7.5.3 How References Differ from Pointers

Aspect	Pointer	Reference
Syntax	Requires * for dereferencing	Used like normal variables
Nullability	Can be `nullptr` or uninitialized	Must refer to a valid object
Reassignment	Can change to point elsewhere	Cannot be reseated
Initialization	Can be declared without initialization	Must be initialized immediately

7.5.4 Using References to Pass Function Parameters

References are commonly used to pass arguments **by reference** to functions, allowing the function to modify the caller's variable without copying.

```cpp
void increment(int& value) {
    value++;  // modifies the original argument
}

int main() {
    int num = 5;
    increment(num);
    std::cout << num << "\n";  // Outputs 6
}
```

Here, `value` is a reference to `num`, so the increment changes `num` directly.

7.5.5 Example: Reference Usage

Full runnable code:

```cpp
#include <iostream>

int main() {
    int original = 100;
    int& alias = original;  // alias is a reference to original

    std::cout << "original: " << original << "\n";  // 100
    std::cout << "alias: " << alias << "\n";        // 100

    alias = 200;  // modifies original through alias

    std::cout << "original after modification: " << original << "\n";  // 200

    return 0;
}
```

7.5.6 Summary

- References are aliases for existing variables, declared with `type&`.
- They provide simpler syntax and guarantee to always refer to a valid object.
- Ideal for function parameters when you want to avoid copying but maintain easy syntax.
- Unlike pointers, references cannot be null or reassigned.

7.6 Pointer to Pointer and Reference to Reference

In C++, pointers and references provide ways to access variables indirectly. This section explores **multi-level indirection** with pointers and explains why **references to references** are not supported.

7.6.1 Pointer to Pointer: Multi-level Indirection

A **pointer to a pointer** is a pointer variable that holds the address of another pointer. This enables multiple layers of indirection, useful in complex data structures and dynamic memory management.

7.6.2 Declaring Pointer to Pointer

```
type** ptrPtr;
```

Here, `ptrPtr` is a pointer to a pointer to a `type`.

7.6.3 Example: Pointer to Pointer

```
int value = 42;
int* pValue = &value;     // pointer to int
int** ppValue = &pValue;  // pointer to pointer to int

std::cout << "Value: " << value << "\n";                // 42
std::cout << "Dereferenced once: " << *pValue << "\n"; // 42
std::cout << "Dereferenced twice: " << **ppValue << "\n"; // 42
```

- `pValue` holds the address of `value`.
- `ppValue` holds the address of `pValue`.
- Dereferencing `ppValue` once gives `pValue`; dereferencing twice accesses `value`.

7.6.4 Practical Use Case: Dynamic 2D Arrays

Pointers to pointers are commonly used to create **dynamic multi-dimensional arrays**:

```
int rows = 3;
int cols = 4;

// Allocate array of pointers (each for a row)
int** matrix = new int*[rows];

// Allocate each row
for (int i = 0; i < rows; ++i) {
    matrix[i] = new int[cols];
}

// Assign and print values
matrix[0][1] = 10;
std::cout << "matrix[0][1] = " << matrix[0][1] << "\n";

// Clean up memory
for (int i = 0; i < rows; ++i) {
    delete[] matrix[i];
}
delete[] matrix;
```

Here, `matrix` is a pointer to a pointer of `int`. The outer pointer stores addresses of row arrays, enabling a flexible 2D array structure.

7.6.5 References to References: Not Supported in C

Unlike pointers, **C++ does not allow references to references**. You cannot declare something like:

```
int& &refRef = someRef; // Illegal in C++
```

7.6.6 Why No References to References?

- References are designed as **aliases** for objects, providing a transparent alternative to pointers.
- They must be **initialized once** and cannot be reseated.
- Allowing references to references would add unnecessary complexity without meaningful benefits.
- If multiple levels of indirection are needed, pointers or pointer-to-pointer types are used instead.

7.6.7 Summary

- **Pointer to pointer** (`type**`) enables multiple layers of indirection.
- Useful for dynamic multi-dimensional arrays and complex data structures.
- **References to references** are not allowed in C++ by design.
- For multi-level indirection, use pointers instead of references.

Chapter 8.

Classes and Objects

1. Class Declaration Syntax
2. Data Members and Member Functions
3. Access Specifiers: `public`, `private`, `protected`
4. Constructors and Destructors Syntax
5. Member Initialization Lists
6. `this` Pointer
7. Static Members

8 Classes and Objects

8.1 Class Declaration Syntax

A **class** in C++ is a user-defined data type that encapsulates **data** and **behavior**. It provides a blueprint for creating objects, bundling variables (called *data members*) and functions (called *member functions* or *methods*) together.

8.1.1 Basic Syntax of a Class Declaration

A class declaration begins with the keyword `class`, followed by the class name, a pair of braces `{}` containing member declarations, and ends with a semicolon `;`.

```cpp
class ClassName {
    // Members (data and functions) go here
};
```

8.1.2 Minimal Example: Empty Class

```cpp
class MyClass {
    // No members yet
};
```

This declares a class named `MyClass`. Although empty, you can create objects of this type.

8.1.3 Example: Simple Class with Data Member and Member Function

```cpp
class Person {
public:
    // Data member
    int age;

    // Member function
    void sayHello() {
        std::cout << "Hello! I am " << age << " years old.\n";
    }
};
```

8.1.4 Explanation:

- **Encapsulation:** The class groups related data (`age`) and behavior (`sayHello()`) into a single entity.
- **Access specifiers:** The keyword `public:` here makes members accessible from outside the class. (More on this in the Access Specifiers section.)
- **Class members** can be variables or functions.
- The class declaration ends with a semicolon `;` — this is required!

8.1.5 Creating Objects from Classes

Once declared, you can create instances (objects) of a class:

```
Person p;
p.age = 30;
p.sayHello();  // Output: Hello! I am 30 years old.
```

8.1.6 Summary

- Use the `class` keyword followed by the class name and braces `{}` to declare a class.
- Classes encapsulate data and functions into a single type.
- Class declarations must end with a semicolon.
- Objects are instances created from classes and have their own copies of members.

8.2 Data Members and Member Functions

A C++ class contains two primary types of members:

- **Data Members**: Variables that hold the state or attributes of an object.
- **Member Functions**: Functions (methods) that define the behavior or actions an object can perform.

Together, these members encapsulate both the data and the operations on that data.

8.2.1 Data Members

Data members represent the attributes or properties of the class. Each object of the class has its own copy of these variables.

Declaring Data Members

Data members are declared inside the class body, typically near the top.

```
class Rectangle {
public:
    int width;   // data member
    int height;  // data member
};
```

Each `Rectangle` object will store its own `width` and `height`.

8.2.2 Member Functions

Member functions define actions that objects of the class can perform. They can access and modify the object's data members.

Defining Member Functions

Member functions are declared inside the class. They can be defined inside the class body or outside it using the scope resolution operator `::`.

```
class Rectangle {
public:
    int width;
    int height;

    // Member function defined inside class
    int area() {
        return width * height;
    }

    // Member function declaration only
    void setSize(int w, int h);
};

// Member function definition outside class
void Rectangle::setSize(int w, int h) {
    width = w;
    height = h;
}
```

8.2.3 Using Data Members and Member Functions

```
#include <iostream>

int main() {
    Rectangle rect;

    // Access data members
    rect.width = 5;
    rect.height = 10;

    // Call member functions
    std::cout << "Area: " << rect.area() << "\n";  // Outputs: Area: 50

    rect.setSize(3, 4);
    std::cout << "New area: " << rect.area() << "\n";  // Outputs: New area: 12

    return 0;
}
```

8.2.4 Summary

- **Data members** store an object's state and are declared as variables inside the class.
- **Member functions** define an object's behavior and operate on data members.
- Member functions can be defined **inside** the class or **outside** using `ClassName::FunctionName`.
- Objects access data members and call member functions using the dot operator ..

8.3 Access Specifiers: `public`, `private`, `protected`

Access specifiers in C++ control the **visibility** and **accessibility** of class members (data members and member functions). They define which parts of your code can access or modify the members of a class, helping enforce **encapsulation** and protect data integrity.

8.3.1 The Three Access Specifiers

Specifier	Accessible From
`public`	Anywhere (inside or outside the class)
`private`	Only within the class itself
`protected`	Within the class and derived classes (inheritance)

8.3.2 Default Access Level

- In a **class**, members are **`private` by default** if no specifier is given.
- In a **struct**, members default to **`public`**.

8.3.3 Using Access Specifiers in Classes

```cpp
class Sample {
public:
    int publicVar;    // accessible anywhere

private:
    int privateVar;   // accessible only inside Sample

protected:
    int protectedVar; // accessible inside Sample and subclasses

public:
    void setValues(int p, int priv, int prot) {
        publicVar = p;
        privateVar = priv;
        protectedVar = prot;
    }

    void printValues() {
        std::cout << "publicVar: " << publicVar << "\n";
        std::cout << "privateVar: " << privateVar << "\n";
        std::cout << "protectedVar: " << protectedVar << "\n";
    }
};
```

8.3.4 Example: Accessing Members

```cpp
int main() {
    Sample obj;

    obj.publicVar = 10;  // OK: public accessible

    // obj.privateVar = 20;  // Error: privateVar is private
    // obj.protectedVar = 30; // Error: protectedVar is protected

    obj.setValues(1, 2, 3);  // OK: public member function can access all members
    obj.printValues();

    return 0;
}
```

8.3.5 Access Specifier Summary

- **`public`** members can be accessed from anywhere.
- **`private`** members are hidden from outside the class and only accessible internally.
- **`protected`** members are like private but accessible in derived classes, enabling controlled inheritance.

- Use access specifiers to **encapsulate** data and expose only necessary parts of the class interface.

8.4 Constructors and Destructors Syntax

In C++, **constructors** and **destructors** are special member functions that manage the lifecycle of objects:

- **Constructors** initialize objects when they are created.
- **Destructors** clean up resources when objects are destroyed.

8.4.1 Constructors

A **constructor** is a member function with the **same name as the class** and **no return type**. It is automatically called when an object is instantiated.

8.4.2 Syntax

```
class ClassName {
public:
    ClassName();                    // Default constructor
    ClassName(int param);           // Parameterized constructor
    ~ClassName();                   // Destructor
};
```

8.4.3 Default Constructor

A constructor without parameters is called a **default constructor**:

```
class Person {
public:
    int age;

    Person() {  // Default constructor
        age = 0;
        std::cout << "Person created with default age 0\n";
    }
};
```

When you create an object:

```
Person p;  // Calls default constructor automatically
```

8.4.4 Parameterized Constructor

Constructors can accept parameters to initialize data members with specific values:

```
class Person {
public:
    int age;

    Person(int a) {  // Parameterized constructor
        age = a;
        std::cout << "Person created with age " << age << "\n";
    }
};
```

Usage:

```
Person p1(25);  // Calls parameterized constructor
```

8.4.5 Destructors

A **destructor** has the same name as the class preceded by a tilde ~ and no parameters or return type. It is called automatically when an object goes out of scope or is deleted, to release resources.

```
class Person {
public:
    ~Person() {
        std::cout << "Person destroyed\n";
    }
};
```

8.4.6 Object Lifecycle Example

```

#include <iostream>

class Person {
public:
    int age;

    Person() {
        age = 0;
        std::cout << "Default constructor called\n";
    }
    Person(int a) {
        age = a;
        std::cout << "Parameterized constructor called with age " << age << "\n";
    }

    ~Person() {
        std::cout << "Destructor called for age " << age << "\n";
    }
};

int main() {
    Person p1;        // Default constructor
    Person p2(30);    // Parameterized constructor

    return 0;         // Destructors called automatically for p1 and p2 here
}
```

Output:

```
Default constructor called
Parameterized constructor called with age 30
Destructor called for age 30
Destructor called for age 0
```

8.4.7 Summary

- **Constructors** initialize objects automatically when created.
- They can be **default** (no parameters) or **parameterized** (with arguments).
- **Destructors** clean up before an object is destroyed.
- Both constructors and destructors have no return types.
- They ensure proper resource management throughout an object's lifecycle.

8.5 Member Initialization Lists

In C++, **member initialization lists** provide a way to initialize data members of a class **before** the constructor's body executes. This method is often more efficient and sometimes necessary, especially for **const members**, **reference members**, and members of classes without default constructors.

8.5.1 Why Use Member Initialization Lists?

- **Direct Initialization:** Members are initialized directly with given values instead of being default-initialized then assigned.
- **Efficiency:** Avoids extra assignment operations.
- **Required for:**
  - `const` data members (must be initialized at declaration).
  - Reference members (cannot be reassigned after initialization).
  - Members of classes that have no default constructor.

8.5.2 Syntax

```
ClassName(parameters) : member1(value1), member2(value2) {
    // constructor body
}
```

8.5.3 Example: Without Member Initialization List (Using Assignment)

```
class Point {
    int x;
    int y;
public:
    Point(int xVal, int yVal) {
        x = xVal;  // assignment, not initialization
        y = yVal;
    }
};
```

In this example, `x` and `y` are **default-initialized** first (with garbage values for primitive types) and then assigned new values inside the constructor body.

8.5.4 Example: Using Member Initialization List

```
class Point {
    int x;
    int y;
public:
    Point(int xVal, int yVal) : x(xVal), y(yVal) {
        // constructor body (can be empty)
    }
};
```

Here, `x` and `y` are **directly initialized** with `xVal` and `yVal` before the constructor body runs.

8.5.5 Member Initialization List with `const` and Reference Members

```
class Example {
    const int constantValue;
    int& refValue;
public:
    Example(int val, int& ref) : constantValue(val), refValue(ref) {
        // Must initialize const and reference members here
    }
};
```

8.5.6 Summary

- Member initialization lists follow the constructor signature after a colon `:`.
- They initialize members before the constructor body executes.
- Using initialization lists is more efficient and necessary for `const`, references, and non-default-constructible members.
- Prefer initialization lists over assignment inside constructors for cleaner, safer, and optimized code.

8.6 `this` Pointer

Inside every non-static member function of a class, C++ provides an implicit pointer named `this`. The `this` pointer **points to the current object** on which the member function is called. It allows the function to refer to the object's own members explicitly.

8.6.1 Purpose of the `this` Pointer

- Access the calling object's members when there is a naming conflict (e.g., between parameters and data members).
- Return the current object from member functions (useful for chaining calls).
- Pass the current object's address to other functions.

8.6.2 How `this` Works

The type of `this` inside a non-const member function is:

```
ClassName* const this;
```

Meaning: a constant pointer to the current object, so you cannot change the pointer itself but can modify the object.

8.6.3 Example 1: Resolving Naming Conflicts

```
class Rectangle {
    int width;
    int height;

public:
    void setSize(int width, int height) {
        // Parameter names shadow data members
        this->width = width;   // 'this->width' refers to data member
        this->height = height; // 'this->height' refers to data member
    }

    int area() {
        return width * height;
    }
};
```

Without `this->`, assignments like `width = width;` would assign the parameter to itself, leaving the data member unchanged.

8.6.4 Example 2: Returning `*this` for Method Chaining

Full runnable code:

```
#include <iostream>
class Counter {
    int count;

public:
    Counter() : count(0) {}

    // Increment count and return current object
    Counter& increment() {
        count++;
        return *this;  // Dereference 'this' pointer to return object reference
    }

    int getCount() {
        return count;
    }
};

int main() {
    Counter c;
    c.increment().increment(); // Calls increment twice using method chaining
    std::cout << c.getCount(); // Outputs 2
}
```

8.6.5 Summary

- The **`this` pointer** implicitly points to the current object inside non-static member functions.
- Use `this->member` to clarify access when names overlap (e.g., parameters vs. members).
- Returning `*this` enables **method chaining** by returning a reference to the object itself.
- The `this` pointer is a fundamental part of object-oriented design in C++.

8.7 Static Members

In C++, **static members** belong to the class itself rather than to any particular object instance. This means **all objects of the class share the same copy** of a static data member, and static member functions can be called without an object.

8.7.1 Static Data Members

- Declared with the keyword `static` inside the class.
- Only one shared instance exists regardless of how many objects are created.
- Must be **defined and initialized outside the class** (unless they are `const` integral types with inline initialization in C++17 and later).

8.7.2 Syntax Example: Static Data Member

Full runnable code:

```
class Counter {
public:
    static int count;  // Declaration of static data member

    Counter() {
        count++;  // Increment shared count whenever an object is created
    }

    static int getCount() {
        return count;  // Static member function accessing static data
    }
};

// Definition and initialization outside the class
int Counter::count = 0;

#include <iostream>

int main() {
    std::cout << "Initial count: " << Counter::getCount() << "\n";

    Counter c1;
    Counter c2;
    Counter c3;

    std::cout << "Count after creating 3 objects: " << Counter::getCount() << "\n";

    return 0;
}
```

Output:

```
Initial count: 0
Count after creating 3 objects: 3
```

8.7.3 Static Member Functions

- Declared with the `static` keyword.
- Can be called without an object using `ClassName::functionName()`.
- Cannot access non-static members directly because they do not belong to any instance.

8.7.4 Example: Static Member Function

Full runnable code:

```
#include <iostream>

class MathUtils {
public:
    static int square(int x) {
        return x * x;
    }
};

int main() {
    std::cout << "Square of 5 is " << MathUtils::square(5) << "\n";
}
```

8.7.5 Summary

- **Static data members** are shared across all objects of a class.
- They must be defined outside the class with the syntax: `Type ClassName::member = value;`.
- **Static member functions** can be called without an object and only access static members.
- Use static members to maintain class-wide state or utility functions that do not depend on individual objects.

Chapter 9.
Operator Overloading

1. Syntax for Overloading Operators as Member or Non-member
2. Overloading Arithmetic, Comparison, Assignment, and Stream Operators
3. Overloading Function Call Operator `operator()`

9 Operator Overloading

9.1 Syntax for Overloading Operators as Member or Non-member

C++ allows you to **overload operators** to provide custom behavior for user-defined types (classes and structs). Operator overloading makes objects behave more like built-in types, improving readability and expressiveness.

There are two main ways to overload operators:

- **Member function overloads:** Operators overloaded as member functions of a class.
- **Non-member function overloads:** Operators overloaded as free functions or friend functions outside the class.

9.1.1 Member Function Overloads

When you overload an operator as a **member function**, the operator acts on the current object (`*this`) as the **left-hand operand**. The right-hand operand (if any) is passed as a parameter.

Syntax:

```cpp
class ClassName {
public:
    ReturnType operatorOp(const OtherType& rhs);  // 'Op' is operator symbol like +, -, ==, etc.
};
```

- Member operator functions take one fewer parameter than the operator's arity because the left operand is the implicit object.
- For **unary operators** (e.g., `++`, `--`, unary `-`), no parameters are needed.
- For **binary operators** (e.g., `+`, `-`, `==`), one parameter represents the right operand.

Example: Member Overload for Addition Operator +

```cpp
class Vector {
    int x, y;
public:
    Vector(int a, int b) : x(a), y(b) {}

    Vector operator+(const Vector& rhs) {
        return Vector(x + rhs.x, y + rhs.y);
    }
};
```

Usage:

```
Vector v1(1, 2), v2(3, 4);
Vector v3 = v1 + v2;  // Calls v1.operator+(v2)
```

9.1.2 Non-member Function Overloads

Operators can also be overloaded as **non-member functions** (either free functions or declared `friend` inside the class for access to private members).

- Non-member overloads take **all operands as parameters**.
- Useful for operators where the left operand is **not a class instance** (e.g., `ostream << object`).
- Required when the left operand is not an object of the class (e.g., for symmetry or implicit conversions).

Syntax:

```
ReturnType operatorOp(const ClassName& lhs, const ClassName& rhs);
```

or as a friend function inside a class:

```
class ClassName {
    friend ReturnType operatorOp(const ClassName& lhs, const ClassName& rhs);
};
```

Example: Non-member Overload for Output Stream Operator <<

```
#include <iostream>

class Vector {
    int x, y;
public:
    Vector(int a, int b) : x(a), y(b) {}

    friend std::ostream& operator<<(std::ostream& os, const Vector& v);
};

std::ostream& operator<<(std::ostream& os, const Vector& v) {
    os << "(" << v.x << ", " << v.y << ")";
    return os;
}
```

Usage:

```
Vector v(5, 10);
std::cout << v << "\n";  // Calls operator<<(std::cout, v)
```

9.1.3 When to Use Member vs Non-member Overloads

Operator Type	Preferred Overload Type	Reason
Operators with left operand class	Member function	Access to private members, natural syntax
Operators where left operand isn't class	Non-member or friend function	To allow implicit conversions on left operand
Stream insertion/extraction (<<, >>)	Non-member or friend function	Left operand is stream object

9.1.4 Summary

- **Member operator functions** have the syntax `ReturnType operatorOp(Param)`, where `*this` is the left operand.
- **Non-member operator functions** take both operands as parameters and can be declared `friend` to access private members.
- Choose the overload form based on operand types and desired implicit conversions.

9.2 Overloading Arithmetic, Comparison, Assignment, and Stream Operators

Operator overloading lets you define how operators behave with your user-defined types. This section covers common operators including arithmetic (`+`, `-`), comparison (`==`), assignment (`=`), and stream insertion/extraction (`<<`, `>>`).

9.2.1 Overloading Arithmetic Operators (+, -, *, /)

Arithmetic operators usually return a new object representing the result without modifying operands.

Example: Overloading + and - as Member Functions

```
class Vector {
    int x, y;

public:
    Vector(int xVal, int yVal) : x(xVal), y(yVal) {}

    // Addition
    Vector operator+(const Vector& rhs) const {
        return Vector(x + rhs.x, y + rhs.y);
    }

    // Subtraction
    Vector operator-(const Vector& rhs) const {
        return Vector(x - rhs.x, y - rhs.y);
    }
};
```

Usage:

```
Vector v1(1, 2), v2(3, 4);
Vector v3 = v1 + v2;  // v3 is (4, 6)
Vector v4 = v2 - v1;  // v4 is (2, 2)
```

Best Practice: Make these operators `const` since they don't modify the object.

9.2.2 Overloading Comparison Operators (==, !=, >, <)

Comparison operators typically return a `bool`. It's common to overload `==` and `!=` in pairs to keep logical consistency.

Example: Overloading == and != as Non-member Functions

```
class Vector {
    int x, y;

public:
    Vector(int xVal, int yVal) : x(xVal), y(yVal) {}

    friend bool operator==(const Vector& lhs, const Vector& rhs) {
        return lhs.x == rhs.x && lhs.y == rhs.y;
    }

    friend bool operator!=(const Vector& lhs, const Vector& rhs) {
        return !(lhs == rhs);
    }
};
```

9.2.3 Overloading Assignment Operator (=)

Assignment operator is special. If your class manages resources (like dynamic memory), you often need to define or disable it explicitly to avoid shallow copying.

Example: Custom Assignment Operator

```
class Buffer {
    int* data;
```

```cpp
    int size;

public:
    Buffer(int s) : size(s), data(new int[s]) {}

    // Copy assignment operator
    Buffer& operator=(const Buffer& rhs) {
        if (this != &rhs) { // protect against self-assignment
            delete[] data;  // release old resource
            size = rhs.size;
            data = new int[size];
            for (int i = 0; i < size; i++)
                data[i] = rhs.data[i];
        }
        return *this;
    }

    ~Buffer() {
        delete[] data;
    }
};
```

Note: If your class does not manage dynamic resources, the compiler-generated assignment is often sufficient.

9.2.4 Overloading Stream Insertion (<<) and Extraction (>>) Operators

These operators are usually overloaded as **non-member** friend functions because the left operand is a stream object (`std::ostream` or `std::istream`).

Example: Overloading << for Output

```cpp
#include <iostream>

class Vector {
    int x, y;

public:
    Vector(int xVal, int yVal) : x(xVal), y(yVal) {}

    friend std::ostream& operator<<(std::ostream& os, const Vector& v) {
        os << "(" << v.x << ", " << v.y << ")";
        return os;
    }
};

int main() {
    Vector v(5, 7);
    std::cout << "Vector v: " << v << "\n";  // Outputs: Vector v: (5, 7)
}
```

Example: Overloading >> for Input

```cpp
#include <iostream>

class Vector {
    int x, y;

public:
    friend std::istream& operator>>(std::istream& is, Vector& v) {
        is >> v.x >> v.y;
        return is;
    }
};
```

9.2.5 Common Pitfalls and Best Practices

- **Return by value** for arithmetic operators to avoid unexpected side effects.
- For **assignment operator**, always check self-assignment and manage resources properly (Rule of Three/Five).
- When overloading **comparison operators**, keep them consistent (`==` and `!=`).
- Stream operators must **return the stream** to allow chaining (`std::cout << v1 << v2;`).
- Avoid overloading operators in ways that make code confusing or unexpected.

9.2.6 Summary

Operator	Typical Return Type	Member or Non-member	Notes
Arithmetic (`+`, `-`)	New object	Member	Usually `const` member functions
Comparison (`==`, `!=`)	`bool`	Non-member/friend	Overload pairs to maintain logical consistency
Assignment (`=`)	Reference to `*this`	Member	Implement carefully when managing resources
Stream insertion/extraction (`<<`, `>>`)	Stream reference	Non-member/friend	Left operand is stream, enabling chaining

9.3 Overloading Function Call Operator `operator()`

C++ allows you to overload the **function call operator** `operator()` to make objects behave like functions. This powerful feature enables creating **functors** (function objects) that can be invoked using the usual function call syntax.

9.3.1 What Is `operator()`?

- It is a special operator that lets you use an instance of a class as if it were a function.
- You define the behavior inside the class by providing an `operator()` member function.
- This operator can take any number and types of arguments.
- The object can maintain state, unlike plain functions.

9.3.2 Syntax

```
ReturnType operator()(ParameterList) {
    // function-like behavior here
}
```

9.3.3 Simple Example: A Functor That Adds a Fixed Value

```
#include <iostream>

class Adder {
    int to_add;

public:
    Adder(int value) : to_add(value) {}

    int operator()(int x) {
        return x + to_add;
    }
};

int main() {
    Adder add_five(5);

    std::cout << add_five(10) << "\n";  // Outputs 15
    std::cout << add_five(20) << "\n";  // Outputs 25
}
```

Here, `add_five` is an object that behaves like a function adding 5 to its argument.

9.3.4 Practical Applications of `operator()`

- **Callbacks and event handlers:** Objects can be passed as callable handlers storing internal state.
- **Custom predicates:** Use functors in algorithms requiring callable conditions.
- **Function adapters:** Wrap behavior with configurable parameters.
- **Lazy evaluation:** Objects can store computation context for delayed invocation.

9.3.5 Multiple Parameters and Const Correctness

You can overload `operator()` with multiple parameters and declare it `const` if it does not modify the object:

```cpp
class Multiplier {
    int factor;

public:
    Multiplier(int f) : factor(f) {}

    int operator()(int a, int b) const {
        return (a * b) * factor;
    }
};

int main() {
    Multiplier mul(3);
    std::cout << mul(2, 4) << "\n";  // Outputs 24 (2*4*3)
}
```

9.3.6 Summary

- Overloading `operator()` lets objects be called like functions.
- Functors can carry state and be more flexible than plain functions or function pointers.
- Useful in standard library algorithms, event-driven programming, and more.

Chapter 10.

Inheritance and Polymorphism

10 Inheritance and Polymorphism

10.1 Derived Classes and Base Classes Syntax

Inheritance is a fundamental feature of C++ that allows a new class (called a **derived class**) to inherit members (data and functions) from an existing class (called a **base class**). This mechanism promotes **code reuse** and enables **specialization** by extending or customizing base class behavior.

10.1.1 Syntax for Single Inheritance

```cpp
class Base {
    // Base class members
};

class Derived : public Base {
    // Additional members or overrides
};
```

- The colon `:` indicates inheritance.
- The access specifier (`public` in this example) controls the accessibility of base class members in the derived class.
- `public` inheritance means "is-a" relationship, i.e., a `Derived` **is a** `Base`.

10.1.2 Simple Example: Base and Derived Classes

Full runnable code:

```cpp
#include <iostream>

class Animal {
public:
    void speak() const {
        std::cout << "Animal sound\n";
    }
};

class Dog : public Animal {
public:
    void bark() const {
        std::cout << "Woof!\n";
    }
};

int main() {
    Dog myDog;
    myDog.speak();  // Inherited from Animal
    myDog.bark();   // Defined in Dog
}
```

Output:

```
Animal sound
Woof!
```

10.1.3 Explanation

- `Dog` inherits the `speak()` function from `Animal`.
- `Dog` adds its own function `bark()`.
- An object of type `Dog` can access both `speak()` and `bark()`.
- This shows how inheritance allows **reusing code** from `Animal` and **specializing** it with new behavior.

10.1.4 Summary

- A **base class** defines common data and behavior.
- A **derived class** inherits from the base and may add or modify members.
- The syntax uses `class Derived : access_specifier Base { ... }`.
- Single inheritance is a powerful way to build hierarchical class relationships and promote code reuse.

10.2 Access Control in Inheritance

When a class inherits from a base class, the **access control** of the base class members and the type of inheritance used affect which members are accessible in the derived class and outside code. Understanding these rules is crucial for designing clear and secure class hierarchies.

10.2.1 Access Specifiers Recap

- `public`: Members are accessible from anywhere.
- `protected`: Members are accessible within the class itself and its derived classes.
- `private`: Members are accessible only within the class itself.

10.2.2 Inheritance Types and Their Effects

When declaring a derived class, you specify an **inheritance access specifier** that affects the accessibility of the base class members in the derived class:

```
class Derived : public Base { ... };    // Public inheritance
class Derived : protected Base { ... }; // Protected inheritance
class Derived : private Base { ... };   // Private inheritance
```

10.2.3 How Inheritance Types Change Member Accessibility

Base Class Member	Public Inheritance	Protected Inheritance	Private Inheritance
`public`	remains `public`	becomes `protected`	becomes `private`
`protected`	remains `protected`	remains `protected`	becomes `private`
`private`	inaccessible	inaccessible	inaccessible

- **Private members** of the base class are never accessible directly by the derived class, regardless of inheritance type.

10.2.4 Example Illustrating Access Control Differences

Full runnable code:

```
#include <iostream>

class Base {
public:
    int public_member = 1;
protected:
    int protected_member = 2;
private:
    int private_member = 3;
};

class PublicDerived : public Base {
public:
    void show() {
        std::cout << public_member << "\n";     // OK: public remains public
        std::cout << protected_member << "\n";  // OK: protected remains protected
        // std::cout << private_member << "\n"; // Error: private inaccessible
    }
};

class ProtectedDerived : protected Base {
public:
    void show() {
        std::cout << public_member << "\n";     // OK: public becomes protected, accessible here
        std::cout << protected_member << "\n";  // OK: protected remains protected
        // std::cout << private_member << "\n"; // Error: private inaccessible
    }
};

class PrivateDerived : private Base {
public:
    void show() {
        std::cout << public_member << "\n";     // OK: public becomes private, accessible here
        std::cout << protected_member << "\n";  // OK: protected becomes private
        // std::cout << private_member << "\n"; // Error: private inaccessible
    }
};

int main() {
    PublicDerived pd;
    std::cout << pd.public_member << "\n"; // OK: public remains public

    ProtectedDerived protd;
    // std::cout << protd.public_member << "\n"; // Error: public_member is protected in ProtectedDeriv

    PrivateDerived privd;
    // std::cout << privd.public_member << "\n"; // Error: public_member is private in PrivateDerived

    return 0;
}
```

10.2.5 Explanation

- In `PublicDerived`, `public_member` stays public, so it is accessible via `pd.public_member`.
- In `ProtectedDerived`, `public_member` becomes protected, so it is not accessible outside the class.
- In `PrivateDerived`, `public_member` becomes private, so it is also inaccessible outside.
- Protected members remain accessible inside derived classes but their outside accessibility depends on inheritance type.
- Private base members are never accessible directly by derived classes.

10.2.6 Summary

- The inheritance specifier (`public`, `protected`, `private`) controls how base class members' access is transformed in the derived class.
- Use **public inheritance** to model "is-a" relationships and keep base members accessible as intended.
- Use **protected** or **private inheritance** to restrict base class interface exposure.

- Private members of the base are always hidden from derived classes.

10.3 Virtual Functions and Overriding

One of C++'s most powerful features is **runtime polymorphism**, which allows the program to decide at runtime which function implementation to call based on the actual object type, rather than the pointer or reference type. This is achieved through **virtual functions**.

10.3.1 Declaring Virtual Functions

A **virtual function** is a member function declared with the `virtual` keyword in a base class. It signals that derived classes may provide their own implementations that override the base class version.

```cpp
class Base {
public:
    virtual void speak() const {
        std::cout << "Base speaking\n";
    }
};
```

10.3.2 Overriding Virtual Functions in Derived Classes

Derived classes can provide their own version of the virtual function by overriding it:

```cpp
class Derived : public Base {
public:
    void speak() const override { // override keyword signals intent
        std::cout << "Derived speaking\n";
    }
};
```

- The `override` keyword (introduced in C++11) tells the compiler that this function overrides a virtual function in the base class, helping catch errors like mismatched signatures.

10.3.3 Dynamic Dispatch and Polymorphism

When calling a virtual function through a **base class pointer or reference**, the version corresponding to the actual object type is invoked — this is called **dynamic dispatch**.

```cpp
void makeSpeak(const Base& obj) {
    obj.speak();
}

int main() {
    Base b;
    Derived d;

    makeSpeak(b); // Outputs: Base speaking
    makeSpeak(d); // Outputs: Derived speaking
}
```

Here, even though `makeSpeak` takes a reference to `Base`, the correct `speak()` function is called according to the actual object's type.

10.3.4 Importance of Virtual Destructors

If a class has virtual functions, its destructor should also be declared `virtual` to ensure proper cleanup when deleting derived objects via base class pointers.

Full runnable code:

```cpp
#include <iostream>
class Base {
public:
    virtual ~Base() {
        std::cout << "Base destructor\n";
    }
};

class Derived : public Base {
public:
    ~Derived() {
        std::cout << "Derived destructor\n";
    }
};

int main() {
    Base* ptr = new Derived();
    delete ptr; // Both Derived and Base destructors called
}
```

Without a virtual destructor, only the base destructor runs, potentially causing resource leaks.

10.3.5 Summary

- **Virtual functions** enable runtime polymorphism by allowing derived classes to override base class functions.
- Use the `virtual` keyword in the base class and the `override` keyword in derived classes for clarity and safety.
- Virtual functions support **dynamic dispatch** when accessed through base pointers or references.
- Always declare destructors as `virtual` in polymorphic base classes to ensure proper cleanup.

10.4 Abstract Classes and Pure Virtual Functions Syntax

In C++, **abstract classes** serve as interfaces or base classes that provide a common contract but are not meant to be instantiated directly. They are defined by including one or more **pure virtual functions**.

10.4.1 What Are Pure Virtual Functions?

A **pure virtual function** is a virtual function declared by assigning `= 0` in its declaration. This indicates that the function has no implementation in the base class and **must** be overridden by any concrete derived class.

```
class Shape {
public:
    virtual void draw() const = 0;  // Pure virtual function
};
```

This declaration means `Shape` is an abstract class.

10.4.2 Characteristics of Abstract Classes

- **Cannot be instantiated:** You cannot create objects of an abstract class.
- **Defines an interface:** Serves as a blueprint for derived classes, enforcing them to implement specific behaviors.
- **May contain implemented member functions:** Abstract classes can have concrete functions and data members as well.

10.4.3 Example: Abstract Class Usage

Full runnable code:

```cpp
#include <iostream>
class Shape {
public:
    virtual void draw() const = 0;  // Pure virtual function

    void describe() const {
        std::cout << "I am a shape." << std::endl;
    }

    virtual ~Shape() = default;  // Virtual destructor
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

int main() {
    // Shape s; // Error: Cannot instantiate abstract class

    Circle c;
    c.draw();        // Output: Drawing a circle.
    c.describe();    // Output: I am a shape.

    Shape* ptr = &c;
    ptr->draw();     // Dynamic dispatch calls Circle::draw()
}
```

- Trying to instantiate `Shape` directly will cause a compilation error.
- `Circle` implements the pure virtual function `draw()`, making it a concrete class.
- A pointer to `Shape` can refer to any derived class instance, supporting polymorphism.

10.4.4 Why Use Abstract Classes?

- Enforce a **contract** for derived classes to implement specific functions.
- Enable **polymorphic behavior** through pointers or references to base class.
- Facilitate **code reuse** and design clarity by defining common interfaces.

10.4.5 Summary

- Declaring a virtual function with `= 0` makes it **pure virtual**.
- Any class with one or more pure virtual functions becomes an **abstract class**.

- Abstract classes **cannot be instantiated** but can be used as base types.
- Derived classes must **override** all pure virtual functions to be instantiable.

10.5 Multiple and Virtual Inheritance Syntax

C++ supports **multiple inheritance**, allowing a derived class to inherit from more than one base class. While powerful, multiple inheritance introduces complexities such as ambiguities and the infamous **diamond problem**.

10.5.1 Multiple Inheritance Syntax

A class can inherit from several base classes by listing them separated by commas:

```cpp
class Base1 {
public:
    void greet() { std::cout << "Hello from Base1\n"; }
};

class Base2 {
public:
    void greet() { std::cout << "Hello from Base2\n"; }
};

class Derived : public Base1, public Base2 {
    // Inherits from both Base1 and Base2
};
```

10.5.2 Ambiguity Problem

Because both `Base1` and `Base2` have a member function named `greet()`, calling `greet()` on `Derived` is ambiguous:

```cpp
int main() {
    Derived d;
    // d.greet(); // Error: ambiguous call to greet()

    d.Base1::greet(); // Calls Base1::greet()
    d.Base2::greet(); // Calls Base2::greet()
}
```

The compiler does not know which base's `greet()` to call unless explicitly qualified.

10.5.3 The Diamond Problem

Multiple inheritance can lead to the **diamond problem** when two base classes share a common ancestor:

```
       Animal
       /    \
  Mammal    Bird
       \    /
      Bat
```

Here, `Bat` inherits from both `Mammal` and `Bird`, which both inherit from `Animal`. Without special handling, `Bat` contains **two copies** of the `Animal` base class.

10.5.4 Virtual Inheritance to the Rescue

To avoid multiple copies of the shared base, C++ uses **virtual inheritance**. It tells the compiler to share the base class between derived classes:

```
class Animal {
public:
    void breathe() { std::cout << "Breathing...\n"; }
};

class Mammal : virtual public Animal { };
class Bird : virtual public Animal { };

class Bat : public Mammal, public Bird { };
```

Now, `Bat` has only **one shared `Animal` base**.

```
int main() {
    Bat b;
    b.breathe();  // Works without ambiguity
}
```

10.5.5 When to Use Multiple and Virtual Inheritance

- **Multiple inheritance** is useful when a class logically **needs to combine behaviors** from multiple base classes.
- Use **virtual inheritance** to solve diamond problem and avoid duplicated bases.
- Be cautious: multiple inheritance can increase complexity, cause ambiguities, and complicate maintenance.

10.5.6 Summary

- Multiple inheritance syntax lists base classes separated by commas.
- Ambiguities arise when base classes share members with identical names.
- The diamond problem occurs when two bases share a common ancestor.
- Virtual inheritance shares the common ancestor, avoiding duplication.
- Use these features judiciously to maintain clear and maintainable code.

Chapter 11.

Template and Generic Programming

1. Function Template Syntax
2. Class Template Syntax
3. Template Specialization and Partial Specialization

11 Template and Generic Programming

11.1 Function Template Syntax

Function templates enable you to write **generic functions** that work with different data types without duplicating code. Instead of writing separate functions for each type, you define a single template function that the compiler can instantiate with specific types as needed.

11.1.1 Template Parameter Syntax

A function template begins with the `template` keyword followed by template parameters enclosed in angle brackets `< >`. The most common template parameter is a **type parameter**, specified with the keyword `typename` or `class` (both are interchangeable in this context):

```cpp
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

Here, `T` is a placeholder type that will be replaced by the actual type when the function is called.

11.1.2 Function Template Instantiation

When you call a template function, the compiler **deduces** the template parameter types from the arguments you provide:

```cpp
int main() {
    int x = 5, y = 10;
    std::cout << max(x, y) << "\n";  // T deduced as int

    double a = 3.14, b = 2.71;
    std::cout << max(a, b) << "\n";  // T deduced as double

    char c1 = 'A', c2 = 'Z';
    std::cout << max(c1, c2) << "\n";  // T deduced as char
}
```

The compiler generates separate versions of `max` for `int`, `double`, and `char` behind the scenes.

11.1.3 Explicit Template Argument Specification

You can also explicitly specify the type when calling the function template:

```
std::cout << max<int>(5, 10) << "\n";
```

But in most cases, automatic type deduction is preferred for brevity.

11.1.4 Practical Example: Generic Swap Function

Templates simplify writing reusable algorithms:

```
template <typename T>
void swapValues(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 1, y = 2;
    swapValues(x, y);  // swaps integers

    double d1 = 3.5, d2 = 4.5;
    swapValues(d1, d2);  // swaps doubles
}
```

Without templates, you'd need separate `swap` functions for each data type.

11.1.5 Summary

- Function templates allow writing **generic functions** with type parameters.
- Template parameters use the syntax `template <typename T>`.
- The compiler deduces types automatically or you can specify them explicitly.
- Templates promote **code reuse** and reduce duplication for similar operations on different types.

11.2 Class Template Syntax

Just like function templates, **class templates** enable you to create **generic classes** that can operate with different data types while maintaining type safety. This is especially useful for implementing reusable containers and data structures without duplicating code for each type.

11.2.1 Declaring a Class Template

A class template begins with the `template` keyword followed by a template parameter list inside angle brackets `< >`. Inside the class, these template parameters can be used as types for data members, function parameters, or return types.

```
template <typename T>
class Box {
public:
    T value;

    Box(T val) : value(val) {}

    T getValue() const {
        return value;
    }

    void setValue(T val) {
        value = val;
    }
};
```

In this example, `T` is a placeholder type that will be replaced with an actual type when an object of `Box` is created.

11.2.2 Using Class Templates: Instantiation

When you create objects of a class template, you specify the actual type for the template parameter:

```
int main() {
    Box<int> intBox(123);
    std::cout << intBox.getValue() << "\n";  // Outputs: 123

    Box<std::string> strBox("Hello");
    std::cout << strBox.getValue() << "\n";  // Outputs: Hello
}
```

The compiler generates separate class definitions for `Box<int>` and `Box<std::string>`, ensuring type safety and optimal code.

11.2.3 Example: Simple Generic Container

Templates are widely used for containers such as stacks, lists, or arrays:

```
template <typename T>
class SimpleStack {
private:
    T data[10];
    int top;

public:
    SimpleStack() : top(-1) {}

    void push(const T& item) {
        if (top < 9)
            data[++top] = item;
    }

    T pop() {
        if (top >= 0)
            return data[top--];
        throw std::out_of_range("Stack is empty");
    }
};
```

This generic stack can hold any data type `T`, avoiding the need to write separate stack classes for each type.

11.2.4 Benefits of Class Templates

- **Type Safety:** Unlike void pointers or generic containers without templates, templates enforce type correctness at compile time.
- **Code Reuse:** Write once, use for any data type.
- **Flexibility:** Works seamlessly with built-in and user-defined types.
- **Performance:** Instantiated templates generate efficient, type-specific code without runtime overhead.

11.2.5 Summary

- Class templates declare generic classes using the syntax `template <typename T> class ClassName`.
- Template parameters can be used for member variables and member functions.
- Instantiating a class template requires specifying the concrete type, e.g., `Box<int>`.
- Class templates power flexible and type-safe generic data structures and algorithms.

11.3 Template Specialization and Partial Specialization

While templates provide great flexibility and generic behavior, sometimes you want to **customize the implementation** for specific types or groups of types. This is where

template specialization comes in.

11.3.1 Full Template Specialization

Full specialization means providing a completely custom definition of a template for a specific type.

Syntax

```cpp
template <>
class Box<bool> {
public:
    bool value;

    Box(bool val) : value(val) {}

    void print() const {
        std::cout << (value ? "true" : "false") << "\n";
    }
};
```

Here, `Box<bool>` is a specialized version of the generic `Box<T>` class customized for the `bool` type.

Example

Full runnable code:

```cpp
#include <iostream>
template <typename T>
class Box {
public:
    T value;
    Box(T val) : value(val) {}
    void print() const {
        std::cout << value << "\n";
    }
};

// Full specialization for bool
template <>
class Box<bool> {
public:
    bool value;
    Box(bool val) : value(val) {}
    void print() const {
        std::cout << (value ? "Yes" : "No") << "\n";
    }
};

int main() {
    Box<int> intBox(42);
    intBox.print();  // Outputs: 42

    Box<bool> boolBox(true);
    boolBox.print(); // Outputs: Yes
}
```

Full specialization is useful when a certain type requires a completely different implementation.

11.3.2 Partial Template Specialization

Partial specialization modifies the template for a subset or pattern of types, without specifying a single exact type.

Note: Partial specialization applies only to **class templates**, not function templates.

Syntax Example: Pointer Partial Specialization

Full runnable code:

```
#include <iostream>
template <typename T>
class Wrapper {
public:
    void info() {
        std::cout << "Generic Wrapper\n";
    }
};

// Partial specialization for pointer types
template <typename T>
class Wrapper<T*> {
public:
    void info() {
        std::cout << "Pointer Wrapper\n";
    }
};

int main() {
    Wrapper<int> w1;
    w1.info();  // Outputs: Generic Wrapper

    Wrapper<int*> w2;
    w2.info();  // Outputs: Pointer Wrapper
}
```

Here, the generic `Wrapper<T>` class is specialized for any pointer type `T*`.

11.3.3 When to Use Specialization

- When the generic implementation does not work efficiently or correctly for specific types.
- To optimize or change behavior for types with unique characteristics (e.g., `bool` or pointers).
- To handle different resource management or interaction patterns for particular data types.

11.3.4 Best Practices

- Use full specialization sparingly to avoid excessive code duplication.
- Prefer partial specialization for broader reuse and cleaner code.
- Always provide a generic fallback template to ensure default behavior.
- Document specializations clearly to maintain code readability.

11.3.5 Summary

- **Full specialization** customizes a template entirely for a specific type.
- **Partial specialization** customizes a template for a subset or pattern of types.
- Partial specialization is limited to class templates, not functions.
- Specialization allows fine-grained control and optimization in generic programming.

Chapter 12.

Exception Handling

12 Exception Handling

12.1 try, catch, and throw Syntax

Exception handling in C++ provides a mechanism to detect and respond to runtime errors or unexpected conditions in a controlled way. The key keywords involved are `try`, `catch`, and `throw`.

12.1.1 The try Block

The `try` block contains code that might throw an exception. If an exception occurs inside the `try` block, control is immediately transferred to the matching `catch` block.

```
try {
    // Code that may throw an exception
}
```

12.1.2 The throw Statement

The `throw` statement is used to signal that an exceptional condition has occurred. It "throws" an exception object, which can be of any type, typically a class or primitive type describing the error.

```
throw exception_object;
```

12.1.3 The catch Block

A `catch` block immediately follows the `try` block and specifies the type of exception it can handle. When an exception is thrown, the first matching `catch` block is executed.

```
catch (ExceptionType e) {
    // Code to handle the exception
}
```

Multiple `catch` blocks can be used to handle different exception types.

12.1.4 How Exceptions Propagate

If an exception is thrown but no matching `catch` block is found in the current function, the exception propagates up the call stack until it is caught or terminates the program.

12.1.5 Simple Example: Throwing and Catching Exceptions

Full runnable code:

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

int divide(int a, int b) {
    if (b == 0) {
        throw std::string("Division by zero error");
    }
    return a / b;
}

int main() {
    try {
        int result = divide(10, 0);
        std::cout << "Result: " << result << "\n";
    }
    catch (const std::string& e) {
        std::cout << "Caught exception: " << e << "\n";
    }

    return 0;
}
```

Explanation:

- `divide` throws a `std::string` exception when division by zero occurs.
- The `main` function calls `divide` inside a `try` block.
- The `catch` block catches the thrown `std::string` and prints an error message.

12.1.6 Summary

- Use `try` blocks to enclose code that may throw exceptions.
- Use `throw` to signal an error condition.
- Use `catch` blocks to handle exceptions based on their types.
- Exceptions propagate up the call stack if not caught immediately.

12.2 Custom Exception Classes Syntax

While C++ allows throwing exceptions of any type, it is a best practice to create user-defined exception classes that provide clear, meaningful error information. Custom exceptions improve error handling clarity and allow more precise catch blocks.

12.2.1 Creating Custom Exception Classes

The recommended approach is to inherit from the standard exception class hierarchy, especially from `std::exception` or one of its derived classes. This enables compatibility with standard library code and consistent interface usage.

12.2.2 Basic Custom Exception Class

```cpp
#include <exception>
#include <string>

class MyException : public std::exception {
    std::string message;

public:
    explicit MyException(const std::string& msg) : message(msg) {}

    // Override the what() function to provide error details
    const char* what() const noexcept override {
        return message.c_str();
    }
};
```

Explanation:

- Inherit from `std::exception`.
- Store an error message inside the class.
- Override `what()` to return a C-style string describing the error.
- Use `noexcept` to guarantee `what()` does not throw exceptions.

12.2.3 Throwing and Catching a Custom Exception

```cpp
#include <iostream>

void process(int value) {
    if (value < 0) {
        throw MyException("Negative value not allowed");
    }
    std::cout << "Processing value: " << value << "\n";
}

int main() {
    try {
        process(-5);
    }
    catch (const MyException& e) {
        std::cout << "Caught custom exception: " << e.what() << "\n";
    }
    return 0;
}
```

12.2.4 Best Practices for Custom Exceptions

- **Derive from `std::exception`:** Provides a common interface for catching exceptions.
- **Override `what()`:** Give meaningful and descriptive error messages.
- **Use exception-safe code:** Avoid throwing exceptions from destructors or `what()`.
- **Keep exception classes lightweight:** Store minimal necessary information.
- **Consider specific exception hierarchies:** For complex programs, define base exception classes and inherit specialized exceptions for clarity.

12.2.5 Extending Standard Exceptions

You can also extend existing standard exceptions for more specialized errors:

```
#include <stdexcept>

class FileNotFoundException : public std::runtime_error {
public:
    explicit FileNotFoundException(const std::string& filename)
        : std::runtime_error("File not found: " + filename) {}
};
```

12.2.6 Summary

- Custom exceptions enhance error handling by providing clear, descriptive error information.
- Derive from `std::exception` and override `what()` to maintain compatibility.
- Throw instances of your custom classes and catch them to handle errors specifically.

12.3 Stack Unwinding and Exception Safety

When an exception is thrown in C++, the program starts a process called **stack unwinding**. This mechanism ensures that the program cleans up resources properly before transferring control to an exception handler. Understanding stack unwinding and exception safety is crucial for writing robust, maintainable code.

12.3.1 What is Stack Unwinding?

Stack unwinding means the automatic destruction of local objects as the runtime moves backward through the call stack to find a matching `catch` block.

- When an exception occurs, the current function exits immediately.
- All local objects in the current scope are destroyed in reverse order of their creation.
- This destruction process continues up the call stack until a suitable `catch` block is found.
- If no `catch` block handles the exception, `std::terminate()` is called, usually ending the program.

This process helps avoid resource leaks by invoking destructors on objects that manage resources like memory, file handles, or locks.

12.3.2 Example of Stack Unwinding

Full runnable code:

```cpp
#include <iostream>
#include <fstream>

void processFile(const char* filename) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        throw std::runtime_error("Cannot open file");
    }

    // Resource cleanup for file is automatic via destructor if exception thrown
    throw std::runtime_error("Error during file processing");
}

int main() {
    try {
        processFile("data.txt");
    }
    catch (const std::exception& e) {
        std::cout << "Exception caught: " << e.what() << '\n';
    }
}
```

- Here, if `processFile` throws an exception, the `std::ifstream` destructor is automatically called, closing the file safely during stack unwinding.

12.3.3 RAII: Resource Acquisition Is Initialization

RAII is a C++ programming idiom that ties resource management to object lifetime:

- Resources (e.g., memory, locks, file handles) are acquired during object construction.
- Resources are released automatically in the destructor.
- This ensures cleanup even during exceptions without requiring explicit code.

Examples include smart pointers (`std::unique_ptr`, `std::shared_ptr`), `std::lock_guard` for mutexes, and standard containers managing dynamic memory.

12.3.4 Exception Safety Guarantees

When writing functions that may throw exceptions, it's helpful to understand different safety levels:

Guarantee	Meaning
No-throw	Function guarantees not to throw any exceptions.
Strong	Function either completes successfully or has no side effects (state remains unchanged).
Basic	Function ensures no resource leaks and keeps the program in a valid state, but changes may persist.

12.3.5 Writing Exception-Safe Code

To achieve these guarantees:

- Use RAII to manage resources automatically.
- Avoid raw pointers and manual `new`/`delete` calls.
- Write operations in a way that either completes fully or rolls back on failure.
- Use `noexcept` to mark functions that do not throw.

12.3.6 Example: Basic vs Strong Guarantee

Full runnable code:

```cpp
#include <vector>
#include <iostream>

void addValueBasic(std::vector<int>& vec, int value) {
    vec.push_back(value);  // Basic guarantee: vector remains valid even if exception occurs
}

void addValueStrong(std::vector<int>& vec, int value) {
    std::vector<int> temp = vec; // Make copy first
    temp.push_back(value);
    vec = temp; // Assign back if all succeeds
}

int main() {
    std::vector<int> data = {1, 2, 3};

    try {
        addValueStrong(data, 4);
        std::cout << "Added value successfully.\n";
    }
    catch (...) {
        std::cout << "Exception occurred, but data unchanged.\n";
    }
}
```

- `addValueBasic` modifies `vec` directly; if `push_back` throws, `vec` may be partially modified.
- `addValueStrong` works on a copy and only updates `vec` if successful, maintaining the strong guarantee.

12.3.7 Summary

- **Stack unwinding** automatically cleans up objects when exceptions propagate.
- **RAII** is essential to ensure resources are released safely during exceptions.
- Understanding **exception safety guarantees** helps you design robust functions.
- Aim to write **no-throw or strong guarantee** functions when possible, using safe coding patterns and standard utilities.

Chapter 13.

Namespaces and Scopes

1. Defining and Using Namespaces
2. The `using` Directive and Declaration
3. Global, Local, and Class Scopes

13 Namespaces and Scopes

13.1 Defining and Using Namespaces

As C++ projects grow in size and complexity, so does the likelihood of **name collisions**—situations where two identifiers (such as variables, functions, or classes) share the same name but serve different purposes. For example, two libraries might each define a function named `print()`, causing a conflict when both libraries are used in the same program.

To address this problem and help **organize code into logical groups**, C++ provides **namespaces**. A namespace is a declarative region that provides a scope to the identifiers it contains. This prevents naming conflicts and clarifies where different parts of code come from.

Declaring a Namespace

A namespace is defined using the `namespace` keyword followed by a block of declarations:

```cpp
namespace Math {
    double pi = 3.14159;

    double square(double x) {
        return x * x;
    }
}
```

In this example, both `pi` and `square` are members of the `Math` namespace.

Accessing Namespace Members

To refer to a name declared inside a namespace, use the **scope resolution operator** (`::`):

```cpp
#include <iostream>

int main() {
    std::cout << "pi = " << Math::pi << '\n';
    std::cout << "square(5) = " << Math::square(5) << '\n';
    return 0;
}
```

Here, `Math::pi` and `Math::square` access the members inside the `Math` namespace.

Nested Namespaces

Namespaces can also be **nested** inside one another to further structure code hierarchically:

```cpp
namespace Company {
    namespace Project {
        void log() {
            std::cout << "Project log" << std::endl;
        }
    }
}
```

Accessing a nested member uses multiple scope resolution operators:

```
Company::Project::log();
```

Starting with C++17, nested namespaces can be written more compactly:

```
namespace Company::Project {
    void log() {
        std::cout << "Project log (C++17 syntax)" << std::endl;
    }
}
```

Namespace Aliasing

To simplify access to deeply nested namespaces or to make code more concise, you can create an **alias** using the `namespace` keyword:

```
namespace CP = Company::Project;

int main() {
    CP::log();  // Equivalent to Company::Project::log()
    return 0;
}
```

Aliases are especially helpful in large projects or when using third-party libraries with long or verbose namespace paths.

By using namespaces effectively, you can keep your code **modular**, **organized**, and **safe from naming conflicts**, especially when integrating code from multiple sources or libraries.

In the next section, we'll explore how the `using` directive and declaration can make working with namespaces more convenient—while also discussing the trade-offs.

13.2 The using Directive and Declaration

When working with namespaces in C++, you often need to refer to identifiers declared within them. While the **scope resolution operator (`::`)** allows precise access to these members, constantly prefixing names can make code verbose. C++ offers two features to simplify this: the **`using` directive** and the **`using` declaration**. Though similar in appearance, they have **very different implications** for name resolution and code safety.

`using namespace`: The Using Directive

The `using namespace` directive makes **all names** in a namespace available in the current scope. This can be convenient but may introduce **naming conflicts** or **pollute the global namespace**.

Syntax:

```
using namespace std;
```

This directive allows you to omit the `std::` prefix when using standard library names:

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

int main() {
    cout << "Hello, world!" << endl; // std::cout and std::endl now available unqualified
    return 0;
}
```

Pitfalls:

Using the directive in global scope—especially in headers or widely shared code—can lead to subtle bugs:

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

void log(string message) {
    cout << message << endl;  // Assumes std::string
}
```

If another library also defines a `string` type, the compiler may become confused about which `string` is meant, especially as more `using namespace` directives are added.

Best Practice:

- **Avoid** `using namespace` in global scope.
- **Never** use it in header files.
- Use it **locally within functions or blocks** where its impact is limited.

`using`: The Using Declaration

In contrast, a `using` declaration **brings in a specific name** from a namespace, avoiding the global pollution associated with the directive.

Syntax:

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

This allows you to refer to `cout` and `endl` without `std::`:

Full runnable code:

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

int main() {
    cout << "Using declaration example." << endl;
    return 0;
}
```

Here, only `cout` and `endl` are brought into the current scope. All other names in `std` remain unaffected. This improves clarity and reduces the chance of name clashes.

Best Practice:

- Use `using` declarations in implementation files or functions to improve readability.
- Prefer them over `using namespace` for safety and precision.

Comparing the Two

Feature	`using namespace` Directive	`using` Declaration
Scope Pollution	High — imports entire namespace	Low — imports only specific names
Risk of Name Collision	High	Low
Readability	Can be clutter-free but risky	Clear and controlled
Use in Headers	**Never recommended**	Acceptable if carefully limited
Ideal Usage	Local scopes for short-lived utility	Most code, especially functions

Summary

- The **`using` directive** (`using namespace`) brings in *all* names from a namespace, which can lead to ambiguity and conflicts.
- The **`using` declaration** (`using std::cout`) imports *specific names*, offering a safer and more precise alternative.
- Favor **declarations** for clarity and maintainability.
- Use **directives** only in **limited, local scopes**, and never in headers.

In the next section, we'll explore how C++ handles different levels of **scope**—from global to local to class-based—and how names are resolved within these regions.

13.3 Global, Local, and Class Scopes

In C++, **scope** determines where an identifier (such as a variable, function, or class) is **visible** and **accessible**. Understanding the different kinds of scope is essential for writing clear, correct code and avoiding subtle bugs like **name shadowing** or unintended access to global variables.

C++ defines several scopes, the most important being:

- **Global scope**
- **Local scope (function/block)**
- **Class (or member) scope**

Let's examine each one in detail.

Global Scope

A name declared **outside of all functions, classes, and namespaces** resides in the **global scope**. It is visible from the point of declaration to the end of the translation unit (usually the entire .cpp file), unless hidden by a local declaration.

Full runnable code:

```cpp
#include <iostream>

int globalCount = 10;  // Global scope

void printCount() {
    std::cout << "Global count: " << globalCount << std::endl;
}

int main() {
    printCount();  // Accesses the globalCount variable
    return 0;
}
```

Global variables exist **throughout the lifetime of the program**. While they can be useful for shared state, overusing them can lead to code that is hard to debug and maintain.

Local Scope

A name declared inside a **function**, **loop**, **conditional block**, or even a compound statement ({} block) is said to be in **local scope**. Its visibility is limited to the block in which it is declared.

```cpp
void example() {
    int localValue = 42;  // Local scope
    std::cout << "Local value: " << localValue << std::endl;
}
```

Variables in local scope are **created** when their block is entered and **destroyed** when the block is exited. This makes local variables ideal for temporary computations and keeping state limited.

Variable Shadowing

A **local variable can shadow a global one** (or an outer-scope variable with the same name), making the outer variable temporarily inaccessible:

```cpp
int value = 100;  // Global variable

void shadowExample() {
    int value = 200;  // Shadows the global 'value'
    std::cout << "Local value: " << value << std::endl;
}
```

Output:

```
Local value: 200
```

The global value is hidden inside shadowExample(). To access the global variable explicitly, use the **scope resolution operator with no prefix**:

```cpp
std::cout << "Global value: " << ::value << std::endl;
```

Class Scope

When a variable or function is declared inside a class, it is said to be in **class scope**. Members of a class are visible to all member functions of the class (and can be accessed using the `this` pointer in instance methods).

```
class Counter {
private:
    int count;  // Class (member) scope

public:
    Counter() : count(0) {}

    void increment() {
        count++;  // Refers to the class member 'count'
    }

    void print() {
        std::cout << "Count: " << count << std::endl;
    }
};
```

Shadowing in Class Scope

Member variables can be shadowed by local variables or function parameters:

```
class Demo {
private:
    int value;

public:
    void setValue(int value) {
        // 'value' here refers to the parameter, not the member
        this->value = value;  // Use 'this->' to refer to the class member
    }
};
```

Using `this->` disambiguates the member variable from the parameter.

Summary Table

Scope Type	Declared In	Lifetime	Accessible From
Global	Outside all functions/classes	Entire program	From declaration onward
Local	Inside function/block	Block entry to exit	Only within that function or block
Class Member	Inside class definition	Tied to class instances	Member functions, `this` pointer

Understanding scopes helps you manage **name visibility**, **lifetime of variables**, and avoid **conflicts through shadowing**. Writing clear code often means minimizing the visibility of names—keeping variables as local as possible—and being deliberate when accessing or modifying outer-scope values.

Chapter 14.

Conversions and Casting

1. Implicit and Explicit Type Conversion
2. C-Style Casts
3. C++ Casts: `static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`

14 Conversions and Casting

14.1 Implicit and Explicit Type Conversion

In C++, **type conversion** refers to the process of converting a value from one data type to another. This is a common operation in programming and occurs in many contexts—from arithmetic operations to function calls. Type conversions can be **implicit**, where the compiler automatically converts types, or **explicit**, where the programmer requests the conversion using a cast.

Understanding how and when conversions occur is essential to writing correct, efficient, and bug-free code.

Implicit Type Conversion

An **implicit conversion** is automatically performed by the compiler when it determines that a value of one type is being used in a context that requires a different but compatible type.

This often occurs in the following situations:

- **Arithmetic operations** between different numeric types
- **Function calls** where arguments are passed to parameters of different types
- **Assignment statements** where the variable type differs from the expression type

Example:

```cpp
int a = 5;
double b = 2.5;
double result = a + b;  // 'a' is implicitly converted to double
```

Here, the integer `a` is implicitly converted to a `double` so it can be added to `b`. This is safe because no information is lost in the conversion from `int` to `double`.

Another example:

```cpp
char c = 'A';
int ascii = c;  // Implicitly converts char to int
```

In this case, the character `'A'` is implicitly converted to its ASCII value (65).

Caution with Implicit Conversions Not all implicit conversions are harmless. Some can lead to **loss of precision**, **unexpected behavior**, or **ambiguity**.

```cpp
int x = 10;
int y = 3;
double ratio = x / y;  // Integer division first, result is 3, then converted to double
```

Even though `ratio` is a `double`, both `x` and `y` are integers, so integer division occurs first, and the result `3` is then converted to `3.0`. This might be surprising to the programmer.

Fix:

```cpp
double ratio = static_cast<double>(x) / y;  // Correctly performs floating-point division
```

Explicit Type Conversion (Casting)

An **explicit conversion** (also called a **cast**) tells the compiler to convert a value from one type to another, even if it wouldn't do so automatically. This is necessary when:

- The conversion is **unsafe** or **potentially lossy**
- The types are not **implicitly compatible**
- You want to **force** a particular interpretation of a value

C-style cast syntax:

```
int a = 10;
double b = (double)a / 3;  // Forces a to be treated as a double
```

Function-style cast syntax:

```
double b = double(a) / 3;
```

Both styles achieve the same result. However, in modern C++, **C++-style casts** like `static_cast` are preferred for clarity and type safety (covered in Section 3).

Examples Comparing Implicit and Explicit Conversions

Implicit:

```
int value = 42;
float f = value;  // int to float: safe, done implicitly
```

Explicit:

```
double d = 9.876;
int truncated = (int)d;  // Explicit cast: may lose precision
```

Here, the cast drops the decimal part (`.876`), resulting in `truncated` being `9`.

Summary

Type of Conversion	Performed By	Safety	When to Use
Implicit	Compiler	Usually safe	Common type promotions, compatible types
Explicit (cast)	Programmer	May be unsafe	Precision control, overriding compiler behavior

Key Takeaways:

- Implicit conversions make code cleaner but can cause subtle bugs if types are mixed carelessly.
- Use explicit casts when you're converting between **incompatible** types or need to **control precision** or **interpretation**.
- Always be cautious of **data loss**, especially when converting from floating-point to integer, or from larger types to smaller ones.

In the next section, we'll look at **C-style casts**—the traditional casting syntax inherited from C—and discuss why modern C++ prefers safer alternatives.

14.2 C-Style Casts

14.2.1 C-Style Casts

Before the introduction of C++'s safer, more explicit cast operators, programmers used **C-style casts** to convert between data types. These casts are inherited from the C programming language and remain valid in C++, but they come with **significant risks** due to their broad, unchecked behavior.

Syntax of C-Style Casts

There are two common syntaxes for C-style casts in C++:

```
(type)expression          // Traditional C-style
type(expression)          // Function-style cast
```

Both forms perform the same operation. They instruct the compiler to convert `expression` to the specified `type`, regardless of whether it's safe or meaningful.

Examples:

```
int i = 42;
double d = (double)i;       // Converts int to double
char c = (char)100;         // Converts int to char
int* p = (int*)0x12345678; // Converts an integer to a pointer (dangerous!)
```

These casts can perform **any** kind of conversion that is allowed by the compiler, including conversions between unrelated pointer types and between const and non-const values.

Risks of C-Style Casts

C-style casts are **powerful but dangerous**, because they:

1. **Lack type safety** – They can convert between unrelated types without any warning or check.
2. **Are ambiguous** – They combine the behavior of `static_cast`, `reinterpret_cast`, `const_cast`, and `dynamic_cast` into one syntax, making the intent unclear.
3. **Make debugging harder** – Improper or unsafe casts can cause subtle runtime bugs, memory corruption, or undefined behavior.

Example of unsafe C-style cast:

```
const int x = 100;
int* ptr = (int*)&x;  // Casts away const-undefined behavior if modified
*ptr = 200;           // Dangerous: modifies a const object!
```

This compiles, but it breaks the promise that `x` is immutable. It may cause the program to behave unpredictably on different platforms or compilers.

Why C++ Prefers Safer Alternatives

Modern C++ introduces **explicit, purpose-specific cast operators**:

- `static_cast` – for well-defined conversions (e.g., numeric types)
- `reinterpret_cast` – for low-level pointer reinterpretation
- `const_cast` – to add or remove `const`
- `dynamic_cast` – for safe downcasting in polymorphic hierarchies

Each of these casts clearly communicates **intent** and restricts what kinds of conversions are allowed.

C++-style alternative to a C-style cast:

```
double d = static_cast<double>(i);  // Safer and more expressive
```

To remove const safely:

```
int* p = const_cast<int*>(&x);
```

Using the appropriate C++ cast helps the compiler catch invalid conversions and helps readers understand what kind of operation is being performed.

Summary

Feature	C-Style Cast	C++-Style Casts
Syntax	`(type)expression`	`static_cast<type>(expression)` (etc.)
Safety	Low — no checks or restrictions	High — restricted by cast type
Readability	Low — intent is unclear	High — purpose-specific and explicit
Modern C++ Usage	Discouraged (except in low-level code)	Strongly recommended

Best Practice: Avoid C-style casts in modern C++ unless you're writing **low-level system code** where performance and control override safety. Prefer **C++-style casts** for clarity, safety, and maintainability.

In the next section, we'll explore each of the C++ cast operators in detail and show how to use them correctly and effectively.

14.3 C++ Casts: `static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`

Modern C++ introduces **type-safe casting operators** that replace traditional C-style casts with clearer and more controlled alternatives. Each C++ cast serves a specific purpose and restricts conversions to well-defined, appropriate cases.

Using these casts properly improves code safety, readability, and maintainability.

`static_cast`

The `static_cast` is used for **well-defined, compile-time conversions** between compatible types, such as:

- Converting between numeric types (e.g., `int` to `double`)
- Casting pointers in inheritance hierarchies (upcasts and safe downcasts)
- Converting enums to integers and vice versa

Example: Numeric Conversion

Full runnable code:

```
#include <iostream>

int main() {
    int i = 10;
    double d = static_cast<double>(i);  // int to double
    std::cout << d << std::endl;        // Output: 10.0
}
```

Example: Pointer Conversion in Inheritance

```
class Animal {
public:
    virtual void speak() {}
};

class Dog : public Animal {
public:
    void bark() {}
};

int main() {
    Dog d;
    Animal* a = static_cast<Animal*>(&d);  // Upcast: safe
}
```

> WARNING `static_cast` does not perform runtime checks, so **downcasting** (from base to derived) is risky if the actual object is not of the derived type.

`dynamic_cast`

The `dynamic_cast` is used for **safe downcasting** in polymorphic class hierarchies. It performs a **runtime type check** to ensure that the cast is valid. It only works with pointers and references to **polymorphic types** (i.e., classes with at least one `virtual` function).

Example: Safe Downcast with Runtime Check

Full runnable code:

```
#include <iostream>

class Animal {
public:
    virtual void speak() {}
};

class Dog : public Animal {
public:
    void bark() {
        std::cout << "Woof!\n";
    }
};

class Cat : public Animal {};

int main() {
    Animal* a = new Dog;

    Dog* d = dynamic_cast<Dog*>(a);
    if (d) {
        d->bark();  // Safe to call
    }

    Cat* c = dynamic_cast<Cat*>(a);
    if (c == nullptr) {
        std::cout << "Not a Cat!\n";
    }

    delete a;
}
```

YES Returns `nullptr` if the cast fails (when using pointers). NO Throws `std::bad_cast` if the cast fails on a reference.

Use `dynamic_cast` only when type safety is critical and RTTI (Run-Time Type Information) is enabled.

`const_cast`

The `const_cast` is used to **add or remove `const` and `volatile` qualifiers**. It is the only cast that can remove `const`, but doing so should be done with caution.

Example: Removing `const` for Legacy API

Full runnable code:

```
#include <iostream>

void print(char* msg) {
    std::cout << msg << std::endl;
}

int main() {
    const char* message = "Hello";
    print(const_cast<char*>(message));  // Removes const

    return 0;
}
```

> WARNING **Undefined behavior** occurs if you modify an object that was originally declared `const`.

```
const int x = 42;
int* px = const_cast<int*>(&x);
*px = 100;  // NO Undefined behavior!
```

Use `const_cast` **only** when you're certain the object was not originally `const` or when interfacing with non-const APIs that won't modify the data.

`reinterpret_cast`

The `reinterpret_cast` is used for **low-level, implementation-defined conversions**, such as:

- Casting between unrelated pointer types
- Interpreting an integer as a pointer
- Bit reinterpretation (dangerous and platform-dependent)

Example: Casting Between Pointer Types

Full runnable code:

```
#include <iostream>

int main() {
    int value = 123;
    void* vptr = &value;

    int* iptr = reinterpret_cast<int*>(vptr);  // Unsafe but allowed
    std::cout << *iptr << std::endl;

    return 0;
}
```

Example: Casting Integer to Pointer

```
uintptr_t address = 0x12345678;
int* p = reinterpret_cast<int*>(address);  // Dangerous!
```

> WARNING Use `reinterpret_cast` only in **low-level system code** or where required by interfacing with external APIs or memory manipulation.

Summary Table of C++ Casts

Cast Type	Purpose	Safe?	Run-time Check?	Use Cases
`static_cast`	Well-defined conversions between compatible types	YES Yes	NO No	Numeric conversions, up/downcasting (careful)
`dynamic_cast`	Safe downcasting in polymorphic hierarchies	YES Yes	YES Yes	Polymorphism, RTTI, interface discovery
`const_cast`	Add/remove `const` or `volatile` qualifiers	WARN-ING Risky	NO No	Interfacing with legacy code (no modification)
`reinterpret_cast`	Low-level, unsafe reinterpretation	NO Dan-gerous	NO No	Memory manipulation, external interfacing

Best Practices:

- Prefer `static_cast` for safe, compile-time conversions.
- Use `dynamic_cast` for downcasting in polymorphic hierarchies when type safety matters.
- Avoid `const_cast` unless absolutely necessary—and never modify const objects.
- Use `reinterpret_cast` with extreme caution and only when no safer alternative exists.

Chapter 15.

Lambda and Modern Syntax

1. Lambda Expression Syntax and Capture Lists
2. `auto` Keyword and Type Deduction
3. `decltype` and Trailing Return Types
4. `constexpr` Functions and Variables

15 Lambda and Modern Syntax

15.1 Lambda Expression Syntax and Capture Lists

15.1.1 Lambda Expression Syntax and Capture Lists

C++11 introduced **lambda expressions** as a concise way to define **anonymous function objects** (also called *functors*) directly in your code. Lambdas are particularly useful for short, inline functions passed to algorithms or used as callbacks, offering a compact and expressive syntax.

Understanding lambdas involves grasping three key components:

1. **Capture list**
2. **Parameter list**
3. **Function body**

Basic Lambda Syntax

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

All parts except the capture list and body are optional. Here's a simple lambda:

```
auto add = [](int a, int b) {
    return a + b;
};

int result = add(3, 4);  // result = 7
```

- `[ ]` – Capture list (describes external variables used inside the lambda)
- `(int a, int b)` – Parameters, just like a regular function
- `{ return a + b; }` – Function body

You can omit `-> return_type` if the compiler can deduce it from the return statement.

Capture Lists

The **capture list** allows the lambda to use variables from the surrounding scope. Captures can be:

- **By value** (`[=]` or `[x]`) – Copies the variable into the lambda
- **By reference** (`[&]` or `[&x]`) – Refers to the original variable

Example: Capture by Value

```
int x = 10;
auto f = [x]() {
    std::cout << "x = " << x << std::endl;
};
x = 20;
f();  // Outputs: x = 10 (captured copy)
```

Example: Capture by Reference

```cpp
int x = 10;
auto f = [&x]() {
    std::cout << "x = " << x << std::endl;
};
x = 20;
f();  // Outputs: x = 20 (refers to original)
```

Mixed Captures:

```cpp
int a = 1, b = 2;
auto f = [a, &b]() {
    std::cout << "a = " << a << ", b = " << b << std::endl;
};
```

Mutable Lambdas

By default, lambdas that capture by value cannot modify their captured variables. To allow modification of **copied** variables, use the `mutable` keyword:

```cpp
int x = 5;
auto f = [x]() mutable {
    x += 10;  // OK because lambda is mutable
    std::cout << "x = " << x << std::endl;
};
f();        // Outputs: x = 15
std::cout << x << std::endl;  // x is still 5 outside the lambda
```

Common Use Cases for Lambdas

1. Sorting with Custom Comparator

Full runnable code:

```cpp
#include <algorithm>
#include <vector>
#include <iostream>

int main() {
    std::vector<int> nums = {3, 1, 4, 1, 5};
    std::sort(nums.begin(), nums.end(), [](int a, int b) {
        return a > b;  // Sort descending
    });

    for (int n : nums) std::cout << n << " ";
    // Output: 5 4 3 1 1
}
```

2. Filtering Elements

Full runnable code:

```cpp
#include <algorithm>
#include <vector>
#include <iostream>
```

```
int main() {
    std::vector<int> nums = {1, 2, 3, 4, 5};
    int threshold = 3;

    std::for_each(nums.begin(), nums.end(), [threshold](int x) {
        if (x > threshold)
            std::cout << x << " ";
    });
    // Output: 4 5
}
```

3. Capturing External State

```
int count = 0;
std::vector<int> nums = {10, 20, 30};

std::for_each(nums.begin(), nums.end(), [&count](int) {
    ++count;
});

std::cout << "Counted " << count << " elements.\n";  // Output: 3
```

Summary

Element	Description
`[ ]`	Capture list: defines access to outside variables
`( )`	Parameter list: like function arguments
`-> return_type`	Optional return type (inferred if omitted)
`{ }`	Body: the actual code executed by the lambda
`mutable`	Allows modification of captured-by-value variables

Lambda expressions are a powerful feature of modern C++ that make code cleaner and more expressive—especially when used in combination with the Standard Library.

In the next section, we'll explore the **`auto` keyword** and how **type deduction** simplifies declarations without sacrificing type safety.

15.2 auto Keyword and Type Deduction

The `auto` keyword, introduced in C++11 and enhanced in later standards, allows the compiler to **automatically deduce the type** of a variable from its initializer. This simplifies complex type declarations and improves code readability—**without sacrificing type safety**.

Rather than replacing strong typing, `auto` helps you **write clearer and more maintainable code**, especially when working with templates, iterators, lambdas, and other verbose types.

Basic Usage of auto

```
auto x = 42;          // int
auto y = 3.14;        // double
auto s = "hello";     // const char*
```

Here, the compiler deduces the type of each variable based on the right-hand side of the assignment.

auto with Containers and Iterators

One of the most common and valuable uses of `auto` is when working with STL containers and their **iterators**, which often have long, nested types:

Full runnable code:

```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4};

    // Without auto:
    std::vector<int>::iterator it = numbers.begin();

    // With auto:
    auto it2 = numbers.begin();

    std::cout << *it2 << std::endl;  // Output: 1
}
```

The `auto` version is much more concise and easier to read.

auto in Range-Based For Loops

When using range-based `for` loops, `auto` allows clean iteration without specifying the exact type of the element:

```
std::vector<std::string> names = {"Alice", "Bob", "Carol"};

for (auto name : names) {
    std::cout << name << std::endl;
}
```

To avoid unnecessary copying, especially with large objects, use a reference:

```
for (const auto& name : names) {
    std::cout << name << std::endl;
}
```

auto with Lambdas

Lambdas often return anonymous function objects with unreadable types. `auto` makes them easy to declare and pass around:

```
auto greet = [](const std::string& name) {
    std::cout << "Hello, " << name << "!" << std::endl;
};

greet("C++");  // Output: Hello, C++!
```

You can also use `auto` to capture lambda return types:

```
auto square = [](int x) {
    return x * x;
};

int result = square(5);  // result = 25
```

Type Safety

Although `auto` hides the explicit type, it remains **strongly typed** and bound by C++'s type rules. The compiler enforces that the deduced type matches usage expectations:

```
auto value = 42;        // int
// value = "hello";     // NO Error: cannot assign const char* to int
```

Deductions You Should Know

Expression	Deduced Type
`auto x = 10;`	`int`
`auto y = 3.14;`	`double`
`auto z = "abc";`	`const char*`
`auto& r = x;`	reference to `x`
`const auto& cr = x;`	const reference

WARNING Without reference qualifiers (`&` or `&&`), `auto` always deduces **by value**.

When to Use `auto`

YES Use `auto` when:

- The type is **long or complex** (e.g., iterators, lambda expressions)
- The initializer **makes the type obvious**
- You want to **avoid redundancy**

NO Avoid `auto` when:

- You need to **document or enforce** a specific type explicitly
- The initializer is **ambiguous** or doesn't clearly reveal the type

Summary

- `auto` lets the compiler deduce types at compile time, reducing clutter in code.
- It works well with iterators, lambda expressions, range-based loops, and templates.
- Type safety is preserved: deduced types behave as if you had written them explicitly.

- Use `auto` to write more expressive and maintainable code—without hiding important type information.

In the next section, we'll examine **`decltype`** and **trailing return types**, tools that let you inspect and specify types more precisely in complex contexts.

15.3 `decltype` and Trailing Return Types

Modern C++ provides powerful tools for **introspecting and controlling types**, especially when working with **templates**, **lambda expressions**, and **complex expressions**. Two such tools are:

- `decltype` – queries the type of an expression at compile time.
- **Trailing return types** – let you declare the return type *after* the parameter list, enabling return types that depend on the parameter types.

Both features increase flexibility and expressiveness, particularly in generic code.

`decltype`: Inspecting the Type of an Expression

`decltype` determines the **exact type** of a given expression *without evaluating it.* This is especially useful in generic code where the type must match another expression's type exactly.

Basic Syntax:

```
decltype(expression)  // resolves to the type of the expression
```

Examples:

```
int x = 42;
decltype(x) y = x;  // y is of type int

double z = 3.14;
decltype(z * x) w = z * x;  // w is of type double
```

`decltype` preserves **reference** and **const** qualifiers:

```
const int a = 5;
decltype(a) b = a;       // b is const int

int& ref = x;
decltype(ref) another = x;  // another is int&
```

> WARNING Unlike `auto`, which drops references and constness unless specified, `decltype` keeps them.

`decltype` in Templates

`decltype` is essential for **template metaprogramming**, especially when deducing return types that depend on template arguments.

```cpp
template <typename T1, typename T2>
auto add(T1 a, T2 b) -> decltype(a + b) {
    return a + b;
}
```

Here, the return type is determined based on the expression `a + b`.

Trailing Return Types

In C++11 and later, you can place the return type **after** the parameter list using the `->` syntax. This is especially useful when the return type depends on the parameters, and you can't name it easily before the parameters are known.

Syntax:

```cpp
auto function(parameters) -> return_type {
    // function body
}
```

Example:

```cpp
auto max(int a, int b) -> int {
    return (a > b) ? a : b;
}
```

While this example doesn't *require* a trailing return type, the syntax becomes powerful in generic contexts.

Example with Templates and `decltype`:

```cpp
template <typename T1, typename T2>
auto multiply(T1 a, T2 b) -> decltype(a * b) {
    return a * b;
}
```

This lets you write flexible, reusable functions that can operate on many types while preserving the correct return type.

Lambda Expressions with Trailing Return Types

For **complex lambdas**, especially those used in generic algorithms or metaprogramming, trailing return types clarify or constrain the return type:

```cpp
auto divide = [](double a, double b) -> double {
    return a / b;
};
```

Or use `decltype` for deduction:

```cpp
auto adder = [](auto x, auto y) -> decltype(x + y) {
    return x + y;
};
```

> This is helpful when using lambdas in C++14 or later with generic parameters (`auto`), and you need exact return type control.

Summary: `auto` vs. `decltype`

Feature	`auto`	`decltype`
Deduction	Deduces from initializer value	Deduces from an arbitrary expression
Constness	Drops `const` and `&` unless specified	Preserves exact type including `const`/`&`
Usage	Variable declarations, return types	Type inspection, trailing return, templates

Summary: Trailing Return Types

Feature	Description
Syntax	`auto func(...) -> return_type { ... }`
Use Case	Needed when return type depends on parameters
Combined with `decltype`	Enables type-safe generic programming

By combining `decltype` and trailing return types, C++ lets you write **precise, type-safe, and flexible** code—especially when the return type can't be easily stated up front.

15.4 `constexpr` Functions and Variables

The `constexpr` keyword, introduced in C++11 and expanded in later standards, allows you to declare **constants and functions** that can be **evaluated at compile time**. This enables better performance and guarantees correctness by catching errors early.

Using `constexpr`, you can write code that behaves like regular runtime logic, but with the added benefit that it can be evaluated during compilation whenever possible.

Purpose of `constexpr`

The primary goals of `constexpr` are:

- **Enable compile-time computation** of values.
- **Ensure immutability** and stability of constants.
- **Improve performance** by avoiding runtime computation of known values.
- Allow more complex **constant expressions** in contexts like array sizes, template parameters, and switch labels.

`constexpr` Variables

A `constexpr` variable is a constant whose value is known at compile time.

```
constexpr int size = 10;  // Compile-time constant
int arr[size];            // OK: array size is known at compile time
```

You can also use `constexpr` with user-defined types, provided their constructors are also `constexpr`.

Example:

```
constexpr double pi = 3.1415926535;
constexpr int squared = 5 * 5;
```

> WARNING All values used to initialize a `constexpr` variable must themselves be compile-time constants.

`constexpr` Functions

A `constexpr` function is one that can be evaluated at compile time **if called with constant arguments**. It must follow specific rules to be valid:

- It must contain only **one return statement** in C++11 (relaxed in C++14 and later).
- It cannot perform operations that require runtime information (e.g., I/O, dynamic allocation).
- All called functions and operations must themselves be `constexpr` or constant.

Basic Syntax:

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

constexpr int result = square(4);  // result = 16, evaluated at compile time
```

From C++14 onward, `constexpr` functions may include control flow (`if`, `for`, etc.) and multiple statements:

```
constexpr int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; ++i)
        result *= i;
    return result;
}

constexpr int six_fact = factorial(6);  // 720
```

`constexpr` with User-Defined Types

You can write `constexpr` constructors and member functions to create constant expressions with class types.

```
struct Point {
    int x, y;
    constexpr Point(int a, int b) : x(a), y(b) {}
    constexpr int sum() const { return x + y; }
};
```

```
constexpr Point p(3, 4);
constexpr int total = p.sum();  // 7
```

This makes `Point` usable in contexts where compile-time evaluation is required.

Practical Benefits

- **Performance**: Compile-time evaluation avoids computing values at runtime.
- **Correctness**: Errors are caught earlier—e.g., using invalid array sizes or template parameters.
- **Code Clarity**: Declares programmer intent explicitly (this value never changes).

Summary: `constexpr`

Feature	Description
`constexpr` variable	Compile-time constant value
`constexpr` function	Can be evaluated at compile time if given constant inputs
Use Cases	Array sizes, template arguments, precomputed values
Benefits	Performance, correctness, clearer intent

Example: Full Use Case

Full runnable code:

```
#include <iostream>

constexpr int cube(int x) {
    return x * x * x;
}

int main() {
    constexpr int val = cube(3);  // Computed at compile time

    int runtime_val = 5;
    std::cout << cube(runtime_val) << std::endl;  // Computed at runtime

    return 0;
}
```

This flexibility allows you to **reuse the same logic** both at compile time and runtime depending on the input.

With `constexpr`, modern C++ gives you a powerful tool for expressing **efficient, predictable, and safe** computations. It bridges the gap between the **flexibility of functions** and the **efficiency of constants**.

Chapter 16.

Preprocessors and Macros

1. Macro Definitions and Usage (`#define`)
2. Conditional Compilation (`#ifdef`, `#ifndef`)
3. Include Guards and `#pragma once`

16 Preprocessors and Macros

16.1 Macro Definitions and Usage (#define)

In C++, macros are a feature provided by the **preprocessor**, a tool that runs before the actual compilation. The preprocessor handles **macro substitution**, which replaces tokens in your code according to rules specified by `#define` directives.

What is a Macro?

A macro is a **textual substitution** rule. When the preprocessor encounters a macro invocation, it replaces it literally with the macro's definition before the compiler sees the code.

Defining Macros with `#define`

The syntax for defining a macro is:

```
#define NAME replacement_text
```

- `NAME` is the macro identifier (usually written in uppercase by convention).
- `replacement_text` is the literal text that replaces `NAME` wherever it appears.

Examples:

```
#define PI 3.14159
#define MAX_SIZE 100
```

Anywhere `PI` appears, the preprocessor replaces it with `3.14159`.

Macros for Expressions and Simple Code Snippets

Macros can also define expressions or code fragments:

```
#define SQUARE(x) ((x) * (x))
#define PRINT(msg) std::cout << msg << std::endl;
```

- `SQUARE(x)` is a macro function with parameter `x`.
- `PRINT(msg)` expands to a `std::cout` statement.

Usage:

```
int val = 5;
int sq = SQUARE(val);   // Expands to ((val) * (val)) → 25
PRINT("Hello, world!"); // Expands to std::cout << "Hello, world!" << std::endl;
```

How Macro Substitution Works

The preprocessor scans your source code, and whenever it finds an occurrence of a macro name, it replaces it with the macro's definition. This is done **textually**, without any knowledge of C++ syntax or types.

For instance:

```
int x = SQUARE(3 + 2);
```

is replaced by

```
int x = ((3 + 2) * (3 + 2));
```

which evaluates correctly to 25.

Common Pitfalls of Macros

Despite their usefulness, macros have several downsides due to their purely textual nature:

1. **No Type Safety**

Macros do not respect C++ type rules. They simply substitute text, so errors are harder to diagnose.

```
#define DOUBLE(x) (2 * x)

int a = 5;
double b = DOUBLE(a);    // Expands to (2 * a)
```

If used incorrectly:

```
int result = DOUBLE(1 + 2);  // Expands to (2 * 1 + 2) = 4, NOT 6!
```

Because of operator precedence, the expression evaluates incorrectly.

Fix: Use parentheses around macro parameters and the entire expression.

```
#define DOUBLE(x) (2 * (x))
```

2. **Unintended Side Effects**

If a macro parameter has side effects, like function calls or increment operators, the macro may evaluate it multiple times:

```
#define SQUARE(x) ((x) * (x))

int i = 3;
int val = SQUARE(i++);  // Expands to ((i++) * (i++))
```

This causes `i` to be incremented twice unexpectedly.

3. **Debugging Difficulty**

Because macros are replaced before compilation, errors involving macros often point to the expanded code, making debugging confusing.

When to Use Macros

- Defining **constants** before `constexpr` was available.
- Simple, short code snippets where the overhead of a function is unacceptable.
- Conditional compilation and platform-specific code (covered later).

Modern Alternatives

Where possible, prefer **`const` variables**, **`constexpr`**, or **inline functions** over macros for better type safety and debugging support.

```cpp
constexpr double Pi = 3.14159;

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

Summary

Feature	Description
`#define` macro	Textual replacement before compilation
Usage	Constants, expressions, code snippets
Pitfalls	No type checking, multiple evaluation, debugging complexity
Best Practice	Use sparingly; prefer `constexpr` and inline functions when possible

Macros remain a fundamental part of C++ preprocessing but should be used carefully to avoid subtle bugs and maintain code clarity.

In the next section, we will explore **conditional compilation directives** such as `#ifdef` and `#ifndef`, which control code inclusion based on compile-time conditions.

16.2 Conditional Compilation (`#ifdef`, `#ifndef`)

Conditional compilation directives allow you to **include or exclude parts of your code** depending on whether certain macros are defined. This mechanism is handled by the preprocessor before the compilation stage and is crucial for writing **portable, configurable, and debuggable** programs.

What is Conditional Compilation?

Conditional compilation lets you **control which code gets compiled** by testing macro definitions. You can adapt your program based on:

- The target platform or operating system
- Debug or release builds
- Feature toggles and configuration options

Common Directives

- `#ifdef MACRO` Includes the code that follows **if** `MACRO` is defined.
- `#ifndef MACRO` Includes the code that follows **if** `MACRO` is *not* defined.
- `#else` Provides an alternative code block if the condition is false.

- `#endif` Marks the end of the conditional block.

Basic Syntax and Usage

```
#ifdef DEBUG
    std::cout << "Debug mode is enabled." << std::endl;
#endif
```

Here, the message prints only if `DEBUG` is defined, typically via:

```
#define DEBUG
```

or via a compiler flag, e.g., `-DDEBUG`.

Example: Platform-Specific Code

```
#ifdef _WIN32
    std::cout << "Compiling for Windows" << std::endl;
#elif defined(__linux__)
    std::cout << "Compiling for Linux" << std::endl;
#else
    std::cout << "Unknown platform" << std::endl;
#endif
```

This uses `#ifdef` and `#elif` to select code based on predefined macros indicating the platform.

Example: Using `#ifndef` for Default Values

You can provide default values or code if a macro is not defined:

```
#ifndef MAX_BUFFER_SIZE
#define MAX_BUFFER_SIZE 1024
#endif
```

If `MAX_BUFFER_SIZE` was not previously defined, it is set to 1024.

Example: `#ifdef` with `#else`

```
#ifdef DEBUG
    std::cout << "Debug info: variable x = " << x << std::endl;
#else
    // No debug output in release builds
#endif
```

Here, debug output only compiles in debug mode, keeping release builds clean.

Summary of Directives

Directive	Purpose
`#ifdef`	Compile block if macro is defined
`#ifndef`	Compile block if macro is *not* defined
`#else`	Alternate block if the condition is false
`#endif`	End conditional block

Best Practices

- Use conditional compilation to isolate **platform-specific** or **debug-only** code.
- Avoid scattering many conditional directives inside functions — prefer isolating platform-dependent code in separate files if possible.
- Use `#ifndef` guards to prevent duplicate definitions (covered in the next section).

Conditional compilation is a powerful way to adapt your program's behavior and build process without changing source code manually. In the next section, we'll discuss **include guards and `#pragma once`**, techniques that prevent multiple inclusion of header files.

16.3 Include Guards and `#pragma once`

In C++, header files are commonly included in multiple source files, or even multiple times within the same file via indirect includes. This can cause **multiple definition errors** when the compiler sees the same declarations or definitions more than once. To avoid this, **include guards** are essential.

Why Include Guards?

Consider including a header file `myheader.h` in two different `.cpp` files or multiple times in the same `.cpp` file. Without protection, the compiler will process the header multiple times, potentially leading to:

- Duplicate class, function, or variable definitions
- Compilation errors due to redefinition

Include guards prevent these issues by ensuring that the **contents of a header file are included only once** per translation unit.

Traditional Include Guards

The traditional and most portable way to implement include guards is using `#ifndef`, `#define`, and `#endif` directives surrounding the entire header content.

Pattern:

```
#ifndef UNIQUE_HEADER_NAME_H
#define UNIQUE_HEADER_NAME_H

// Header file contents here

#endif  // UNIQUE_HEADER_NAME_H
```

- `UNIQUE_HEADER_NAME_H` is a macro name, usually based on the header filename.
- The first time the header is included, the macro is undefined, so the content is included and the macro is defined.
- Subsequent inclusions see the macro already defined and skip the content.

Example: Traditional Include Guard

```
// myheader.h

#ifndef MYHEADER_H
#define MYHEADER_H

void greet();

#endif  // MYHEADER_H
```

Usage in source file:

```
#include "myheader.h"
#include "myheader.h"  // Safe: second inclusion is ignored

void greet() {
    // Implementation
}
```

`#pragma once` A Simpler Alternative

`#pragma once` is a **non-standard but widely supported** preprocessor directive that instructs the compiler to include the file only once per translation unit. It simplifies include guards by eliminating the need for macro names.

Example:

```
// myheader.h

#pragma once

void greet();
```

This directive is supported by all major compilers (GCC, Clang, MSVC) and can reduce human error in macro naming.

Advantages of `#pragma once`

- Easier to write and maintain (no need for unique macro names).
- Potentially faster compilation since the compiler can skip reading the file after the first inclusion.
- Avoids subtle bugs due to mismatched macro names.

Considerations

- `#pragma once` is not part of the official C++ standard, but its support is widespread and generally safe to use.
- In very rare cases (e.g., complicated network file systems or symbolic links), `#pragma once` may fail to detect duplicates properly, whereas include guards always work.
- Many projects use both for maximum safety, but this is uncommon.

Summary

Method	Description	Portability	Ease of Use
Traditional Guards	`#ifndef` / `#define` / `#endif`	Fully portable	Slightly verbose
`#pragma once`	Single directive	Widely supported	Simple and concise

Include guards and `#pragma once` are crucial for **maintaining clean, error-free builds** in C++ projects with multiple files and dependencies.

Chapter 17.
Modern Syntax Features

17 Modern Syntax Features

17.1 `decltype(auto)` and Trailing Return Types

Modern C++ emphasizes type safety and expressiveness through powerful type deduction mechanisms. Two such features—`decltype(auto)` and **trailing return types**—are essential tools for writing **clear, correct, and generic code**, particularly in templates and forwarding functions.

17.1.1 `decltype(auto)`: Accurate Type Deduction

The `decltype(auto)` syntax combines the **automatic deduction of `auto`** with the **precise type resolution of `decltype`**. While `auto` deduces types based on value semantics (typically removing references and `const` qualifiers), `decltype(auto)` retains the **exact type**, including:

- Reference qualifiers (`&`, `&&`)
- `const` or `volatile` qualifiers

This precision makes `decltype(auto)` ideal in contexts where the **exact type of an expression** must be preserved.

Example: Comparing `auto` and `decltype(auto)`

```
int x = 10;
int& ref = x;

auto a = ref;           // a is int (copy of x)
decltype(auto) b = ref; // b is int& (reference to x)
```

In this example:

- `auto` deduces `int`, discarding the reference.
- `decltype(auto)` deduces `int&`, preserving the reference.

17.1.2 Use Case: Forwarding Return Types

If a function returns the result of another function, `decltype(auto)` ensures the **returned type matches exactly**, especially when dealing with references.

```
int& getGlobal() {
    static int value = 42;
    return value;
}

decltype(auto) forwardGlobal() {
    return getGlobal();  // Returns int&
}
```

Returning by `auto` here would make a copy; returning by `decltype(auto)` preserves the reference semantics.

17.1.3 `decltype(auto)` in Template Functions

Template functions often benefit from `decltype(auto)` when the return type depends on deduced template parameters.

```
template<typename Func, typename Arg>
decltype(auto) call(Func f, Arg&& arg) {
    return f(std::forward<Arg>(arg));
}
```

This forwarding function preserves both the return type and the reference/value category of `f(arg)`.

17.1.4 Trailing Return Types

A **trailing return type** is a syntax that places the return type **after** the function parameter list using the `->` operator. This is useful when the return type depends on the parameters, which may not be fully known before they're parsed.

17.1.5 Syntax

```
auto function(parameters) -> return_type {
    // body
}
```

Trailing return types are **especially useful in templates**, where types often depend on parameters in complex ways.

Example: Template Function with Trailing Return Type

```
template<typename T1, typename T2>
auto multiply(T1 a, T2 b) -> decltype(a * b) {
    return a * b;
}
```

Here, `decltype(a * b)` correctly computes the type of the expression involving two potentially different types.

17.1.6 Combining Both: `decltype(auto)` with Trailing Return Types

You can also use `decltype(auto)` in the trailing return type position when writing functions where exact return type preservation is critical.

```cpp
template<typename T1, typename T2>
auto sum(T1&& a, T2&& b) -> decltype(auto) {
    return std::forward<T1>(a) + std::forward<T2>(b);
}
```

Alternatively, you can use `decltype(expr)` explicitly:

```cpp
template<typename T1, typename T2>
auto sum(T1&& a, T2&& b) -> decltype(std::forward<T1>(a) + std::forward<T2>(b)) {
    return std::forward<T1>(a) + std::forward<T2>(b);
}
```

This gives fine control over complex type-dependent return values, especially when perfect forwarding is involved.

17.1.7 Summary

Feature	Description
`decltype(auto)`	Deduces the **exact type** of an expression, preserving references/const
Trailing return type	Declares return type **after** parameters; useful for dependent types
Combined use	Enables precise, generic, and type-safe function declarations

17.1.8 When to Use

- Use `decltype(auto)` when:
  - You need **exact type deduction**, especially in return types
  - You're forwarding expressions or referencing variables
- Use trailing return types when:
  - The return type **depends on function parameters**
  - You're writing **template functions** or **generic lambdas**

17.2 noexcept Specifier Syntax

In C++, exception safety is a key aspect of writing robust and predictable code. The `noexcept` specifier allows you to **explicitly state whether a function can throw exceptions**, improving both the **clarity** of your intent and enabling **compiler optimizations**.

Introduced in C++11 and enhanced in later standards, `noexcept` serves as both a promise and a constraint.

17.2.1 Why Use noexcept?

1. **Clarifies intent**: Makes it clear whether a function is expected to throw.
2. **Enables optimizations**: Compilers can generate more efficient code when they know exceptions won't be thrown.
3. **Improves exception safety**: Prevents exception propagation from certain functions (e.g. destructors) where throwing would be dangerous.

17.2.2 Basic Syntax

Unconditional noexcept

You can declare a function as `noexcept` to indicate it will **never throw**:

```
void log() noexcept {
    std::cout << "This function never throws.\n";
}
```

If such a function throws at runtime, `std::terminate` will be called.

Conditional noexcept(expr)

In templates and generic code, you can use **conditional `noexcept`** to make the specifier depend on whether an expression can throw:

```
template<typename T>
void safeCall(T func) noexcept(noexcept(func())) {
    func();
}
```

Here, `safeCall` is `noexcept` **only if** `func()` is `noexcept`.

17.2.3 Examples of Usage

Function Declaration with noexcept

```
int getValue() noexcept {
    return 42;
}
```

Lambdas with noexcept

```
auto safeLambda = []() noexcept {
    std::cout << "Safe to call!\n";
};
```

Template Example with noexcept

```
template<typename T>
void process(T&& val) noexcept(noexcept(perform(val))) {
    perform(val);
}
```

This propagates the exception guarantee of `perform(val)` to the `process` function.

17.2.4 Inspecting with noexcept

The `noexcept` operator (like `sizeof` or `decltype`) checks whether an expression **is declared not to throw** and yields a `bool`:

```
void f() noexcept {}
void g() {}

static_assert(noexcept(f()), "f should not throw");
static_assert(!noexcept(g()), "g might throw");
```

This can be used in static assertions and conditional logic.

17.2.5 Behavior on Exception Throwing

If a `noexcept` function **does** throw an exception, the program will **terminate immediately**:

```
void dangerous() noexcept {
    throw std::runtime_error("Unexpected!");
}

int main() {
    dangerous();  // Causes std::terminate to be called
}
```

This helps catch violations early during testing or development.

17.2.6 When to Use `noexcept`

Use Case	Reason
Destructors	Must never throw — exceptions during stack unwinding are fatal
Move constructors/assignments	noexcept improves move semantics and allows use in standard containers
Small utility or logging functions	Clarifies intent and improves optimization
Generic code	Enables conditional exception guarantees using `noexcept(expr)`

17.2.7 Summary

Syntax	Meaning
`noexcept`	Declares function cannot throw exceptions
`noexcept(expr)`	Conditionally declares function based on whether `expr` throws
`noexcept(expr)` (operator)	Returns `true` if `expr` is known not to throw

By using `noexcept`, you communicate important information to the compiler and other programmers, enabling better code generation, clearer contracts, and safer exception handling.

17.3 `alignas`, `alignof` Keywords

C++ offers low-level memory control tools for developers working with performance-critical or hardware-specific code. Among these tools are the keywords `alignas` and `alignof`, introduced in C++11. These keywords give you control over and information about **type alignment**, which affects **how data is stored in memory** and **how efficiently it can be accessed**.

17.3.1 What Is Alignment?

Alignment refers to how data is arranged in memory with respect to boundaries (such as 4-byte or 16-byte boundaries). Most processors fetch data more efficiently when it's aligned to a certain size. Misaligned access can result in **slower performance** or even **runtime errors** on some systems.

17.3.2 `alignof`: Querying Type Alignment

The `alignof` operator returns the **alignment requirement** (in bytes) of a type.

Syntax

```
std::size_t alignment = alignof(Type);
```

Example

Full runnable code:

```
#include <iostream>
#include <cstddef>

int main() {
    std::cout << "Alignment of int: " << alignof(int) << '\n';
    std::cout << "Alignment of double: " << alignof(double) << '\n';
}
```

Typical output might be:

```
Alignment of int: 4
Alignment of double: 8
```

You can also apply `alignof` to custom types:

```
struct MyStruct {
    char c;
    double d;
};

std::cout << "Alignment of MyStruct: " << alignof(MyStruct) << '\n';
```

17.3.3 `alignas`: Specifying Alignment

The `alignas` specifier lets you **request a specific alignment** for a variable or type. This can be useful for:

- **SIMD operations** (e.g., aligning to 16 or 32 bytes for vectorized instructions)
- **Embedded systems** with memory-mapped I/O or hardware buffers
- **Avoiding false sharing** in multi-threaded code

Syntax

```
alignas(N) type variable;
```

- `N` must be a power-of-two integer or the result of `alignof(some_type)`.

Example: Aligning a Variable

```
alignas(16) char buffer[64];  // Aligned to 16-byte boundary
```

This ensures `buffer` starts at a memory address divisible by 16, which is beneficial for some CPU instructions.

Example: Aligning a Struct

```
struct alignas(32) AlignedStruct {
    float data[8];
};
```

Now, any instance of `AlignedStruct` is aligned to a 32-byte boundary.

You can also align individual members:

```
struct Mixed {
    char a;
    alignas(16) int b;
};
```

This forces member `b` to be aligned to 16 bytes, possibly inserting padding before it.

17.3.4 Alignment and Performance

Correct alignment:

- Reduces **CPU cycles** needed to load/store data.
- Enables **SIMD (Single Instruction, Multiple Data)** operations.
- Prevents undefined behavior on hardware that requires strict alignment.

17.3.5 Combining `alignas` and `alignof`

```
alignas(alignof(double)) int aligned_int;  // Align int to match double
```

This can be useful when interfacing with hardware or ensuring compatibility between types.

17.3.6 Summary

Keyword	Purpose	Example
`alignof`	Queries the required alignment of a type	`alignof(int)` → 4
`alignas`	Specifies a required alignment	`alignas(16) float data[4];`
Use Cases	Performance tuning, SIMD, embedded systems	Struct or variable alignment

17.3.7 Best Practices

- Use `alignof` to **inspect** types when writing generic or low-level code.
- Use `alignas` **sparingly and purposefully** — unnecessary alignment increases memory usage.
- Always ensure alignment values are **powers of two** and supported by your target platform.

17.4 Inline Variables and Static Assertions

C++17 introduced two powerful features to make code more maintainable and safer:

- **Inline variables**, which allow global variable definitions in header files without violating the One Definition Rule (ODR)
- **`static_assert`**, which enables compile-time checks for enforcing assumptions and constraints

Both features contribute to **clearer**, **more robust**, and **easier-to-maintain** code.

17.4.1 Inline Variables

Prior to C++17, defining non-`const` global variables in header files would violate the **One Definition Rule**, causing multiple definitions across translation units.

C++17 solves this by introducing **inline variables**, which can be defined in headers safely and included in multiple source files without causing linkage errors.

17.4.2 Syntax

```
inline const int max_value = 100;
```

- The `inline` keyword tells the compiler that this variable **may appear in multiple translation units**, but **should be treated as a single definition**.

17.4.3 Example: Inline Constant in a Header File

```
// config.h
#pragma once

inline constexpr int buffer_size = 1024;
```

```
// main.cpp
#include "config.h"
#include <iostream>

int main() {
    std::cout << "Buffer size: " << buffer_size << '\n';
}
```

This works safely even if `config.h` is included in multiple `.cpp` files.

17.4.4 Use Cases

- Global constants in header files
- Shared configuration values
- Template metaprogramming constants

17.4.5 Notes

- Inline variables must be defined with `inline`, not just `extern`, to avoid multiple definition errors.
- `constexpr` variables defined in headers are implicitly inline since C++17, so `inline` is not required (but still valid).

17.4.6 `static_assert`: Compile-Time Checks

The `static_assert` keyword performs a **compile-time assertion**. If the condition evaluates to `false`, the compiler generates an error with an optional message.

This helps catch errors early by enforcing **type constraints**, **size assumptions**, or **valid template parameters** during compilation.

17.4.7 Syntax

```
static_assert(condition, "Error message");
```

17.4.8 Example: Enforcing Type Size

```
static_assert(sizeof(int) == 4, "Expected 4-byte int");
```

If `sizeof(int)` is not 4 bytes on the platform, this produces a compile-time error.

17.4.9 Example: Validating Template Parameter

```
template<typename T>
void process(T val) {
    static_assert(std::is_integral<T>::value, "T must be an integral type");
    // safe to proceed
}
```

Calling `process("text")` would cause a compile-time error because `const char*` is not an integral type.

17.4.10 Usage with `constexpr` and `if constexpr`

You can use `static_assert` with `constexpr` functions or inside `if constexpr` branches:

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

static_assert(square(3) == 9, "Incorrect square function");
```

This verifies correctness **at compile time**, not just at runtime.

17.4.11 Summary

Feature	Purpose	Example
`inline` variable	Define globals in headers safely	`inline constexpr int x = 10;`
`static_assert`	Compile-time condition checking	`static_assert(sizeof(int) == 4, "Wrong size");`

17.4.12 Best Practices

- Use **inline variables** for shared constants in headers, especially with templates or configuration flags.
- Use **`static_assert`** early in templates or functions to enforce assumptions and prevent cryptic errors.
- Prefer `constexpr` values with `static_assert` to evaluate logic entirely at compile time.