Object Oriented Programming

Course Code: CSE-0613-2203

Md. Zahid Akon

Lecturer

Department of CSE

CLO'S

01 Understand the foundational concepts and syntax of C++ programming.

02 Apply control structures, functions, and modular programming techniques.

03 Implement object-oriented programming concepts like classes, inheritance, and polymorphism

04 Utilize encapsulation, abstraction, and dynamic memory management effectively.

05 Develop reusable and efficient solutions using templates, STL, and exception handling.

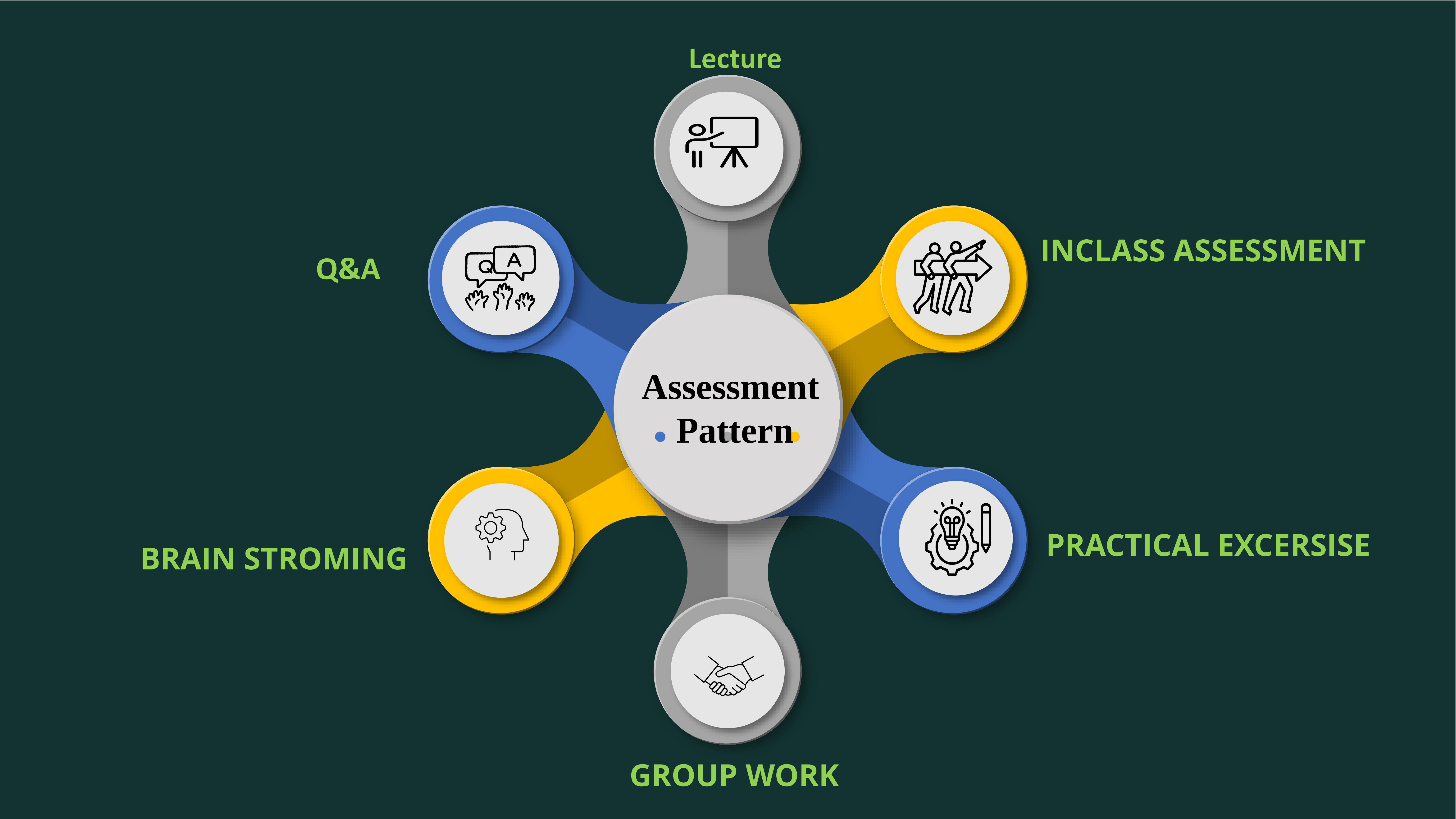
06 Integrate all learned concepts to design, develop, and debug complete projects.

Recommended Books

1. E. Balagurusamy, "Object-Oriented Programming with C++", Tata McGraw-Hill (ISBN: 9781259029936)

2. Bjarne Stroustrup, "The C++ Programming Language", Addison-Wesley (ISBN: 9780321563842)

3. Scott Meyers, "Effective C++", Addison-Wesley (ISBN: 9780321334879)


Lecture
Q&A
INCLASS ASSESSMENT
Assessment
Pattern
BRAIN STROMING
PRACTICAL EXCERSISE
GROUP WORK
Q
A

Course Plan

Week No.	Topics and Key Outcomes	Teaching-Learning Strategies	Assessment Strategies	Alignment to CLO
1	Introduction to C++: Basics of programming, installing tools, writing the first program, variables, and I/O.	Lecture, multimedia, hands-on practice	Feedback, Q&A, simple quiz	CLO1
2	Operators and Control Structures: Using operators, `if` statements, and loops (`for`, `while`).	Lecture, practical examples	Feedback, Q&A, short quiz	CLO2
3	Functions and Arrays: Creating functions, passing values, recursion, and using 1D/2D arrays.	Lecture, hands-on practice	Midterm Quiz #1, practice problems	CLO2
4	Introduction to OOP: Difference between procedural and object-oriented programming, basic class and object.	Lecture, group discussions	Feedback, Q&A	CLO3
5	Classes and Objects: Constructors, destructors, member functions, and `this` pointer.	Lecture, problem-solving sessions	Case Study #1, Assignment #1	CLO3
6	Inheritance: Base/derived classes, types of inheritance, and constructor/destructor chaining.	Lecture, multimedia	Feedback, Q&A, discussions	CLO4
7	Polymorphism: Function overloading, virtual functions, abstract classes, and dynamic method dispatch.	Lecture, group discussions	Feedback, Q&A, examples	CLO4
8	Encapsulation: Grouping data and controlling access with private, protected, and public modifiers.	Lecture, hands-on practice	Feedback, Q&A, quizzes	CLO3
9	Abstraction: Hiding implementation details and designing abstract classes and interfaces.	Lecture, hands-on practice	Feedback, Q&A, quizzes	CLO3

Course Plan

10	**Pointers and Memory**: Working with pointers, `new`/`delete`, and smart pointers.	Lecture, multimedia	Feedback, Q&A, simple assignments	CLO3
11	**File Handling**: Reading/writing files, working with binary files, and random file access.	Lecture, practical examples	Feedback, Q&A	CLO3
12	**Templates**: Creating generic functions and classes using templates.	Lecture, group exercises	Feedback, short quiz	CLO4
13	**Standard Template Library (STL)**: Using vectors, lists, maps, and common algorithms like `sort` and `find`.	Lecture, hands-on practice	Feedback, assignments	CLO4
14	**Exception Handling**: Handling errors with `try`, `catch`, and `throw`; creating custom exceptions.	Lecture, practical examples	Feedback, Q&A	CLO3
15	**Advanced Concepts**: Multiple inheritance, namespaces, and typecasting (`dynamic_cast`, `static_cast`).	Lecture, hands-on exercises	Feedback, practice problems	CLO4
16	**Project Work**: Planning, building, testing, and reviewing a project like Library Management or Banking System.	Group work, instructor guidance	Project reviews, peer evaluations	CLO5
17	**Revision and Final Assessment**: Review of all topics, practical problems, and final exams.	Lecture, problem-solving sessions	Final written and practical exams	CLO5

Week 1

Introduction to C++

Introduction to C++

Welcome to your journey into the world of C++ programming. This presentation provides a foundation in the fundamental concepts that will empower you to build software and solve real-world problems.

Installing C++ Tools and IDE

Compiler

A compiler translates your C++ code into machine-readable instructions. Popular compilers include g++ and clang.

IDE

An Integrated Development Environment (IDE) offers features like code editing, debugging, and project management. Common IDEs include Visual Studio Code, Code::Blocks, and CLion.

Writing Your First C++ Program

Code

```
#include
using namespace std;

int main() {
  cout << "Hello,
World!";
  return 0;
}
```

Output

```
Hello, World!
```

Variables and Data Types

int

Integer values, whole numbers, e.g., 10, -5, 0.

float, double

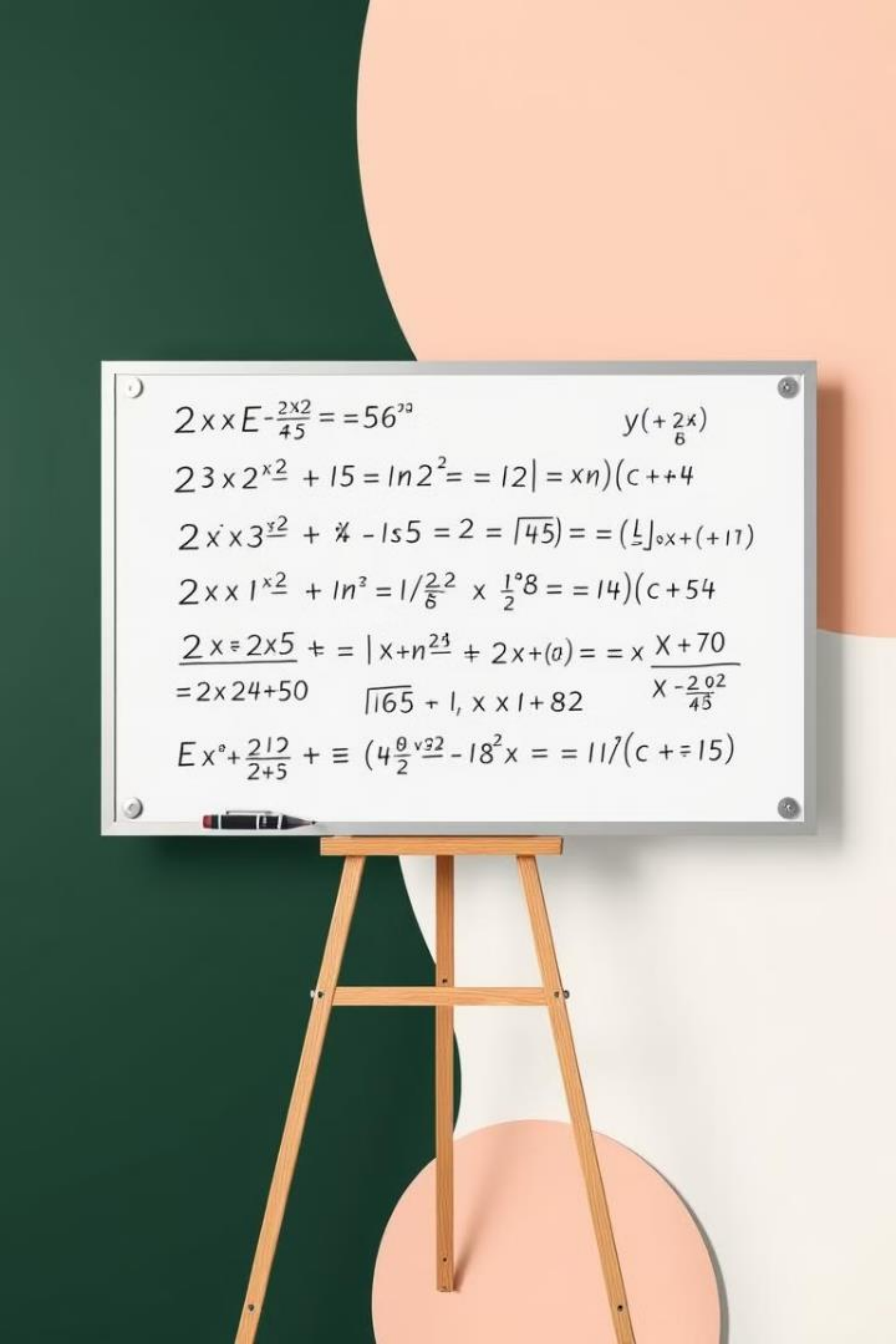
Floating-point numbers, with decimal values, e.g., 3.14, -2.5.

char

Single characters, e.g., 'A', '?', '%'.

string

Sequences of characters, e.g., "Hello", "C++".



Arithmetic Operations and Expressions

1

Addition

Adding two numbers, e.g., 5 + 3 = 8.

2

Subtraction

Subtracting one number from another, e.g., 10 - 7 = 3.

3

Multiplication

Multiplying two numbers, e.g., 2 * 6 = 12.

4

Division

Dividing one number by another, e.g., 15 / 5 = 3.

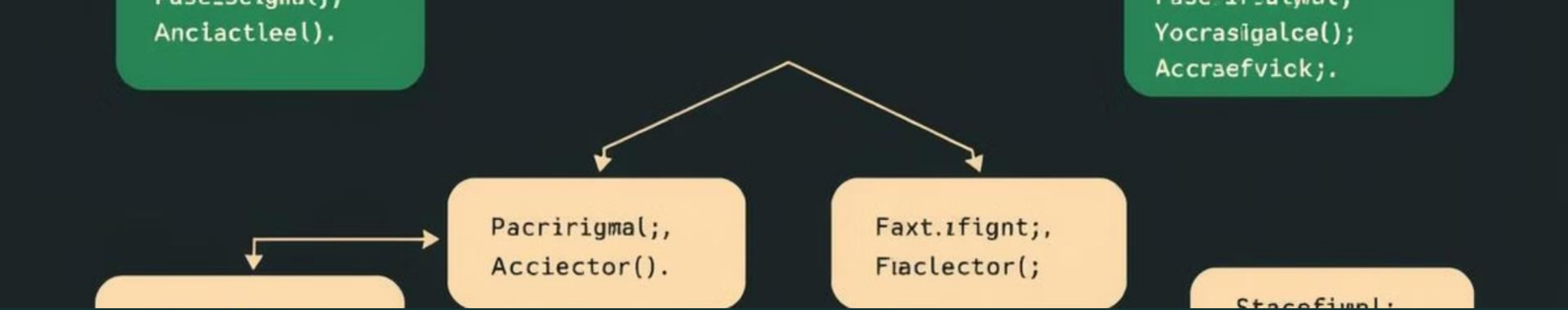
User Input and Output (I/O)

Input

```
#include
using namespace std;

int main() {
  int age;
  cout << "Enter your age: ";
  cin >> age;
  cout << "You are " << age << " years
old.";
  return 0;
}
```

Output

```
Enter your age: 25
You are 25 years old.
```



Control Structures: Conditional Statements

1

2

3

if

Executes a block of code if a condition is true.

else if

Executes a block of code if the previous if condition is false and a new condition is true.

else

Executes a block of code if all previous if and else if conditions are false.

Control Structures: Loops

1. **while**

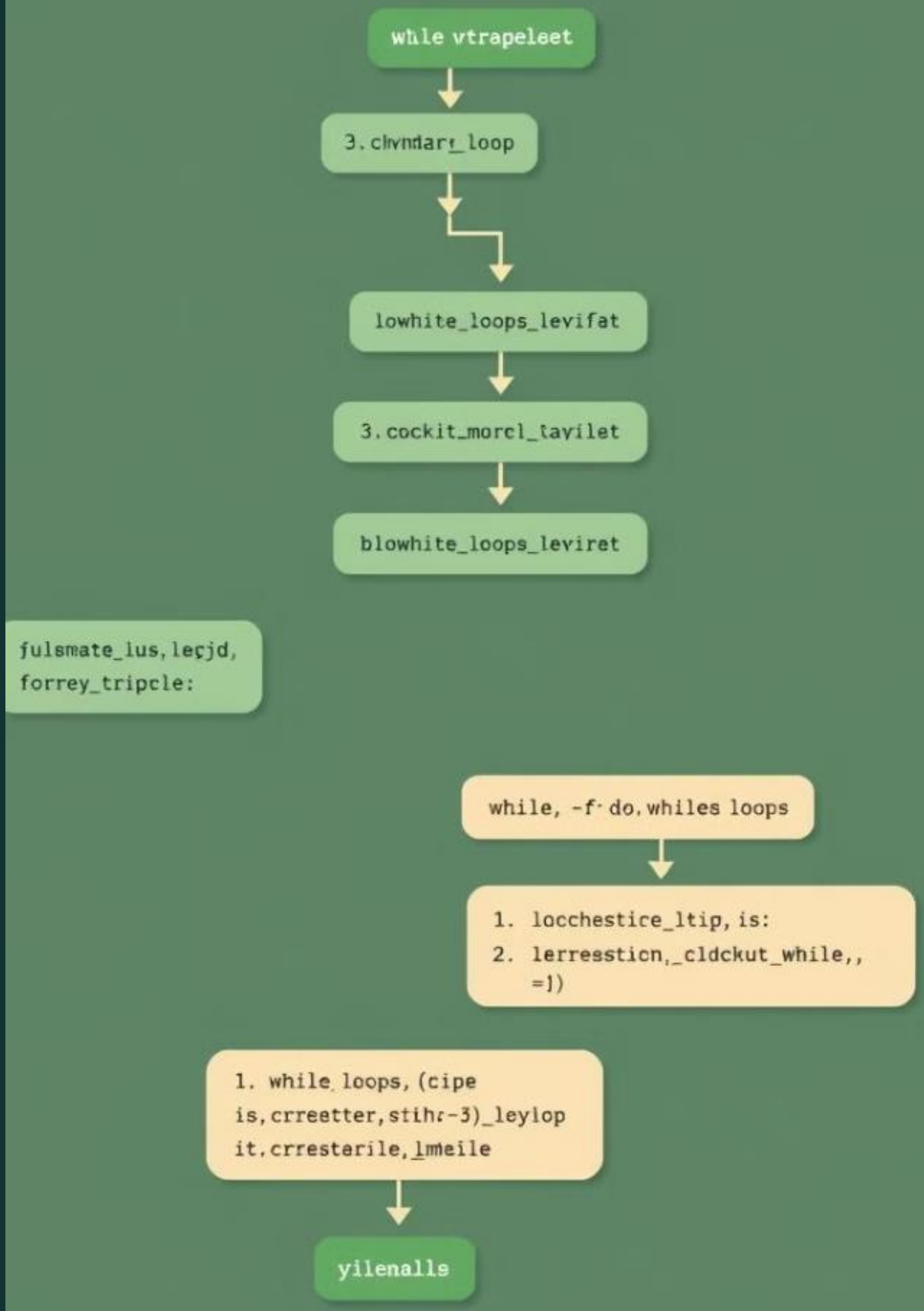
   Executes a block of code repeatedly as long as a condition is true.

2. **for**

   Executes a block of code a specified number of times.

3. **do-while**

   Executes a block of code at least once, and then repeatedly as long as a condition is true.



Functions and Subroutines

1

Function Definition

A function is a block of code that performs a specific task.

2

Function Call

The main program calls a function to execute its code.

3

Parameters

Functions can receive input values through parameters.

4

Return Value

Functions can return a value back to the calling program.

Conclusion and Next Steps

Congratulations! You've mastered the fundamentals of C++ programming. Now explore more advanced concepts such as classes, objects, and data structures. Practice regularly, experiment with new features, and build your skills to become a confident C++ developer.

Week 2
Operators and Control Structures in C++
Introduction to OOP

Operators and Control Structures in C++

This presentation will explore the fundamentals of C++ operators and control structures, essential building blocks for programming.

Types of Operators in C++

Arithmetic Operators

Used for basic mathematical operations.

Relational Operators

Used for comparing values.

Logical Operators

Used for combining logical expressions.

Bitwise Operators

Used for manipulating individual bits.

Arithmetic Operators

Addition (+)

Adds two operands.

Subtraction (-)

Subtracts the second operand from the first.

Multiplication (*)

Multiplies two operands.

Division (/)

Divides the first operand by the second.

Modulo (%)

Returns the remainder of a division.

Relational Operators

Greater Than (>)

Checks if the first operand is greater than the second.

Less Than (<)

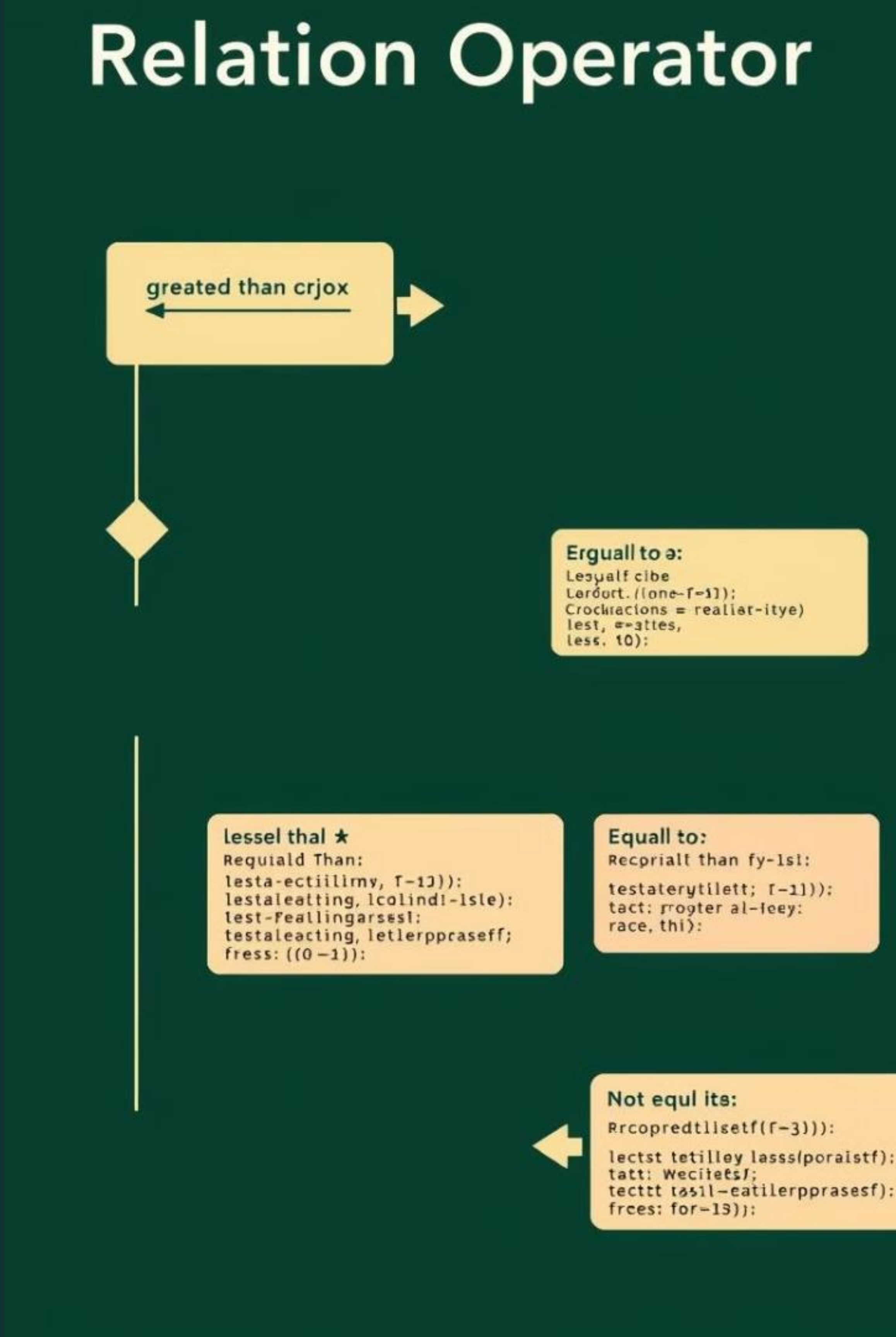
Checks if the first operand is less than the second.

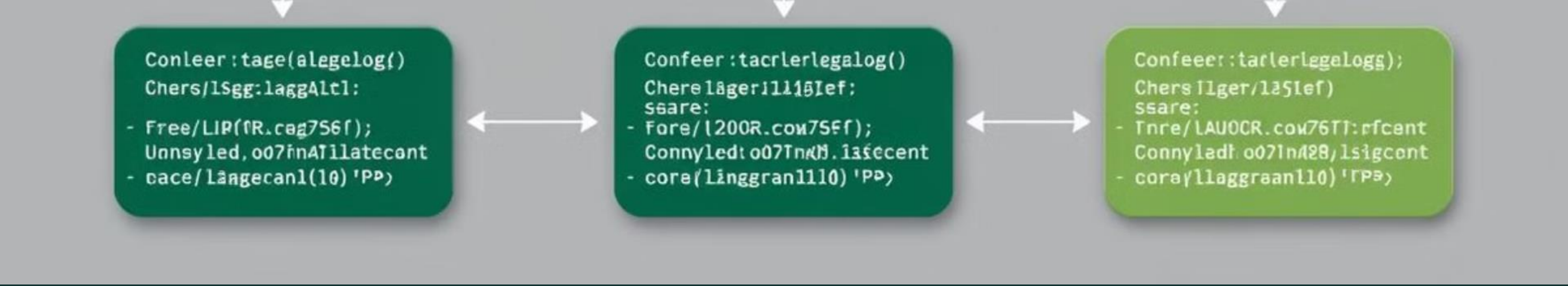
Equal To (==)

Checks if two operands are equal.

Not Equal To (!=)

Checks if two operands are not equal.





Logical Operators

1. Logical AND (&&)

Returns true if both operands are true.

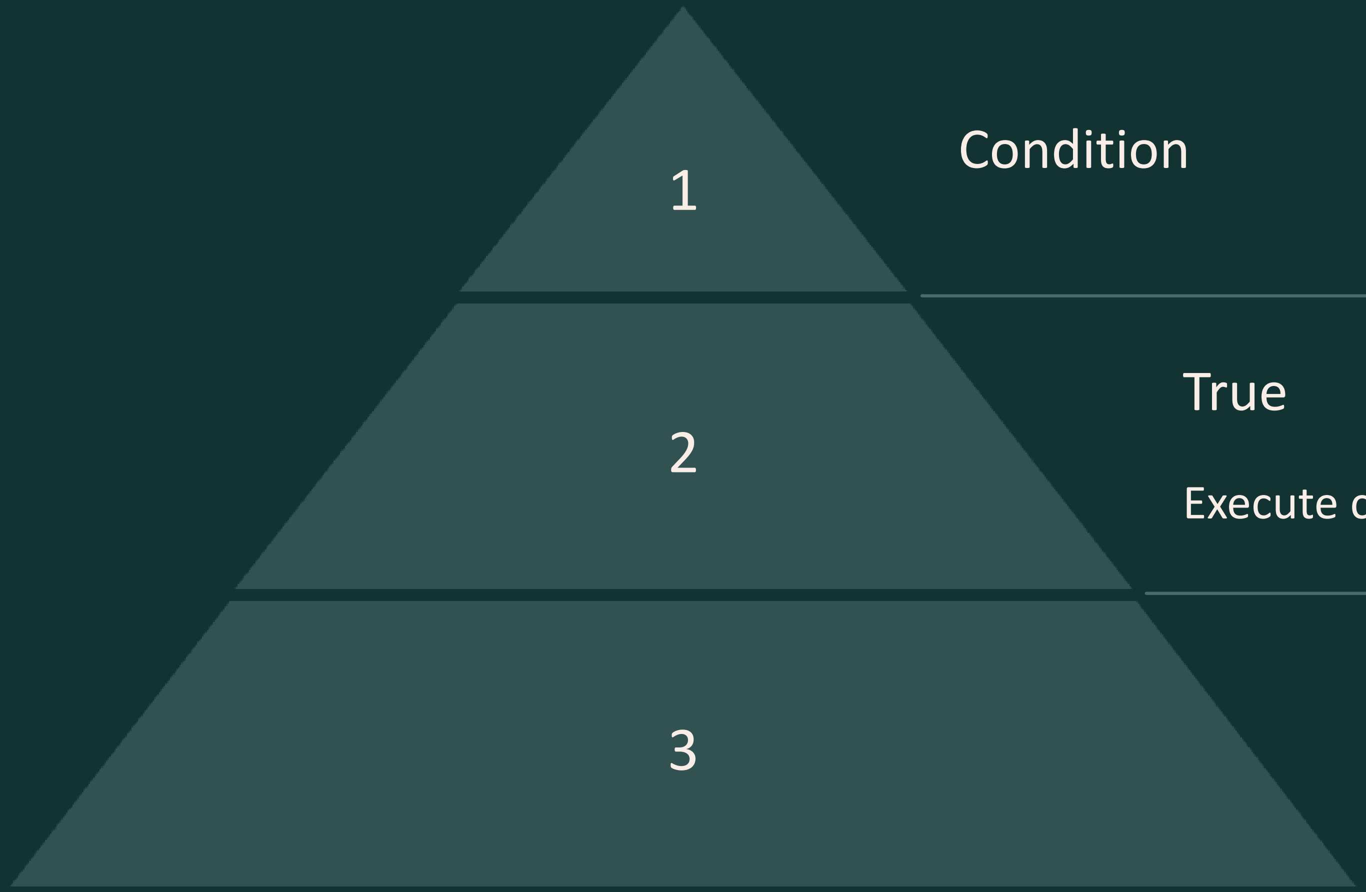
2. Logical OR (||)

Returns true if at least one operand is true.

3. Logical NOT (!)

Reverses the logical state of an operand.

If Statements in C++



Condition

True

Execute code block.

False

Skip code block.

An if statement executes a code block if a condition is true. If false, the block is skipped.

If-Else Statements

1 Condition

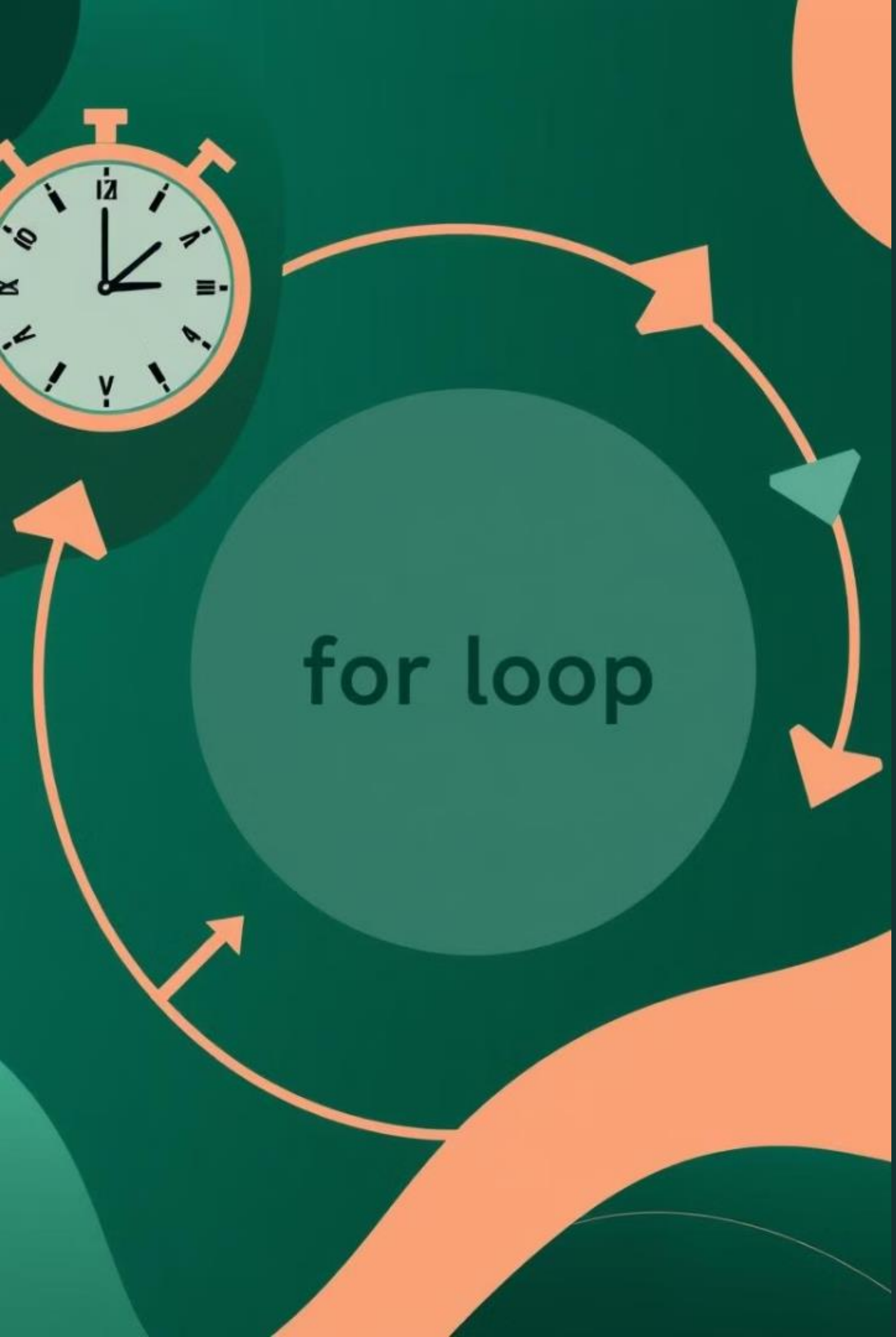
2 True

Execute code block 1.

3 False

Execute code block 2.

If the condition is true, the first code block is executed; otherwise, the second code block is executed.



For Loops in C++

1

Initialization

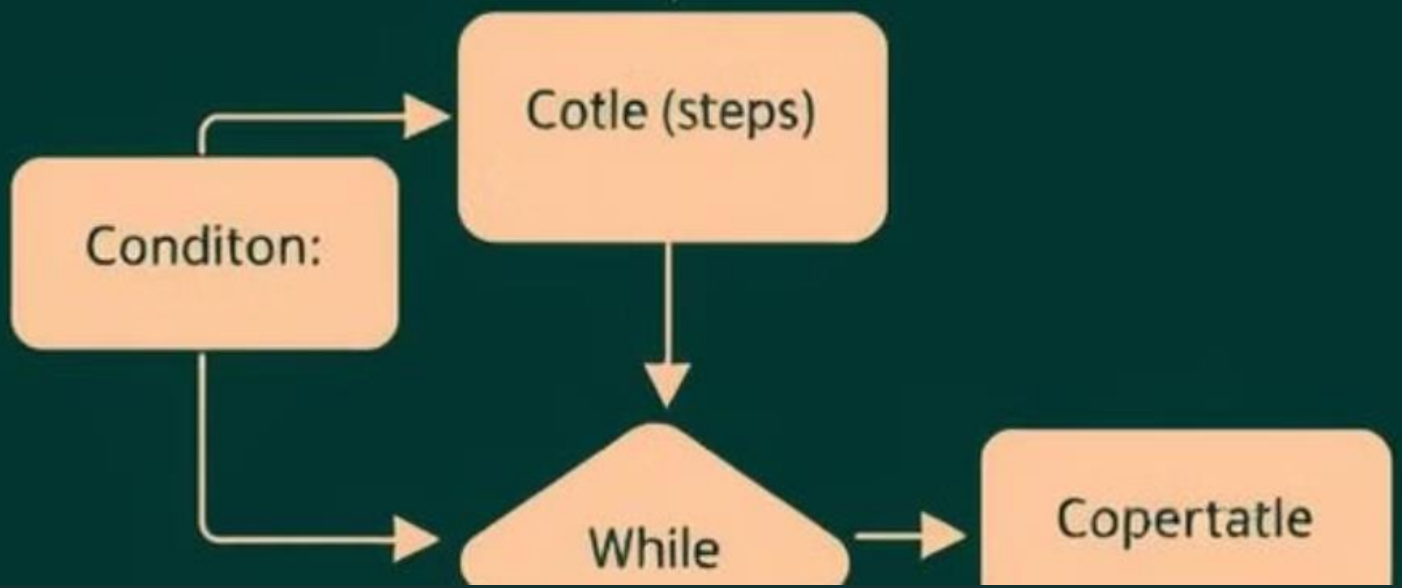
2

Condition

3

Increment

A for loop executes a block of code repeatedly, based on an initialization, condition, and increment/decrement.



While Loops in C++

1	2	3
Condition	True	False
	Execute code block.	Exit loop.

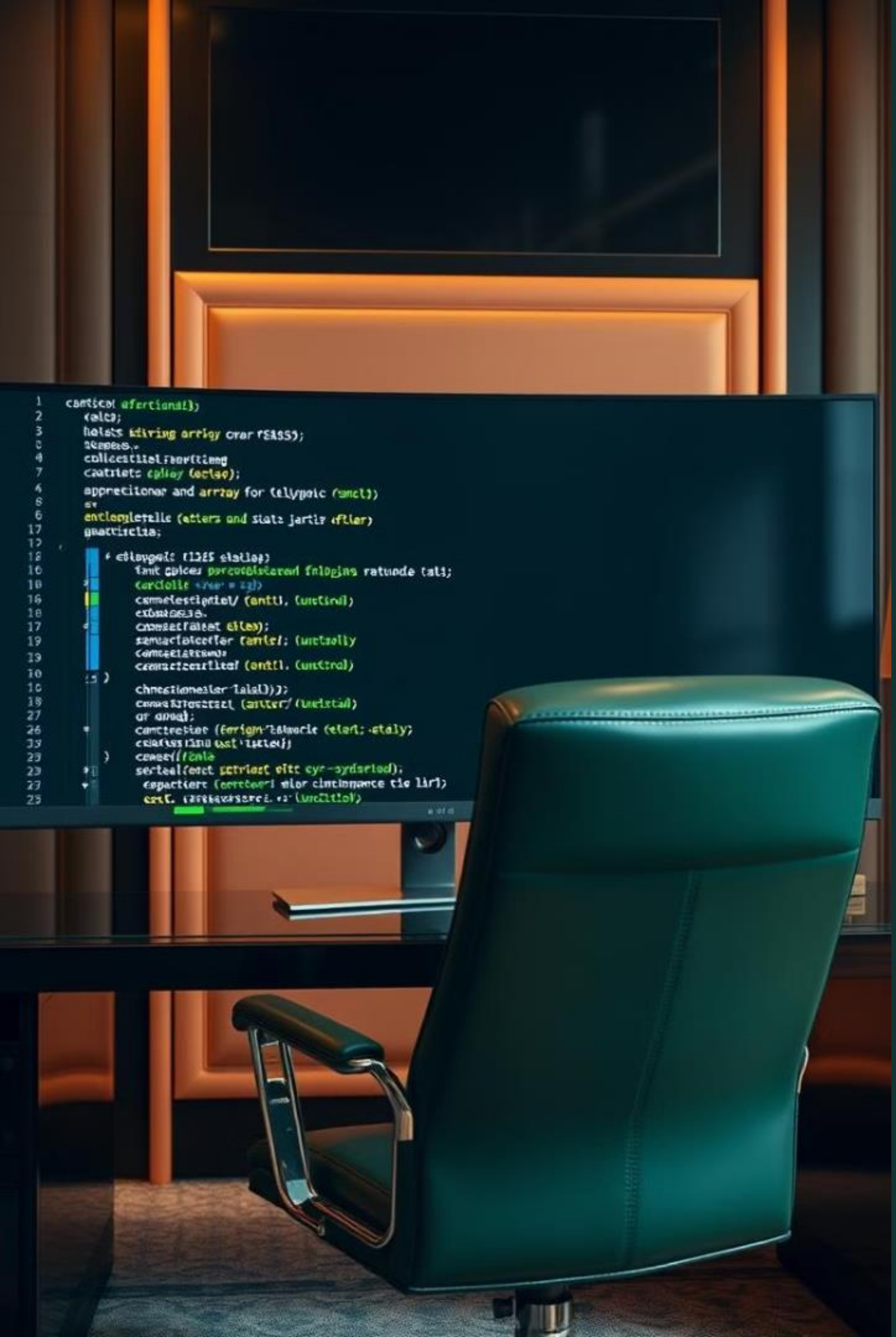
A while loop executes a block of code repeatedly as long as a condition remains true. Once the condition becomes false, the loop terminates.

Conclusion and Key Takeaways

Operators and control structures form the core of programming logic. By mastering these concepts, you can create dynamic and efficient C++ programs. Continue to explore and experiment with these fundamental building blocks to enhance your programming skills.

Week 3

Functions and Arrays in C++



Functions and Arrays in C++

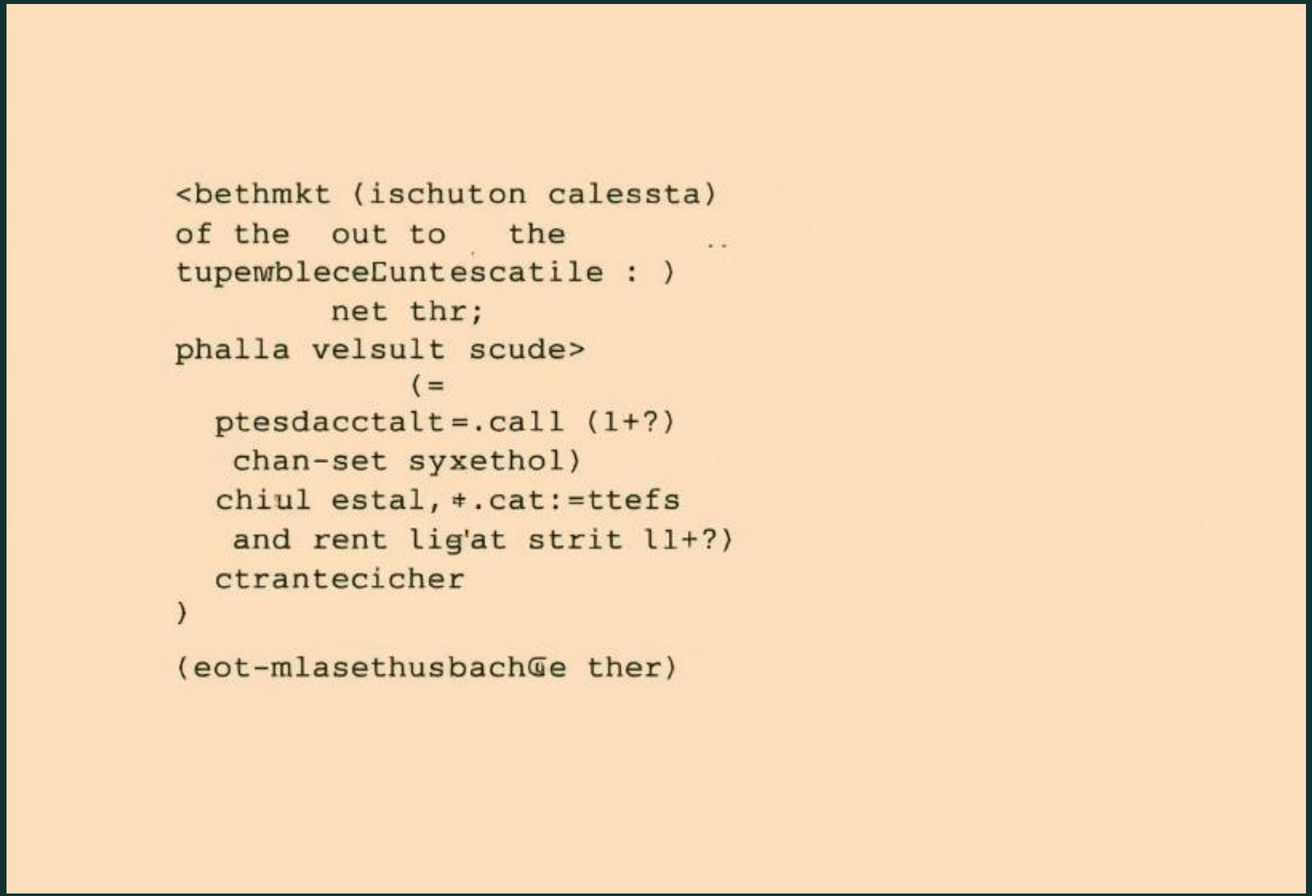
This presentation explores fundamental concepts of functions and arrays in C++, providing a structured overview and practical examples.

Understanding Functions

Functions are reusable blocks of code that perform specific tasks.

They enhance code organization and readability.

They promote modularity and code reusability.



Defining and Calling Functions

Defining a function involves specifying its name, return type, and parameters.

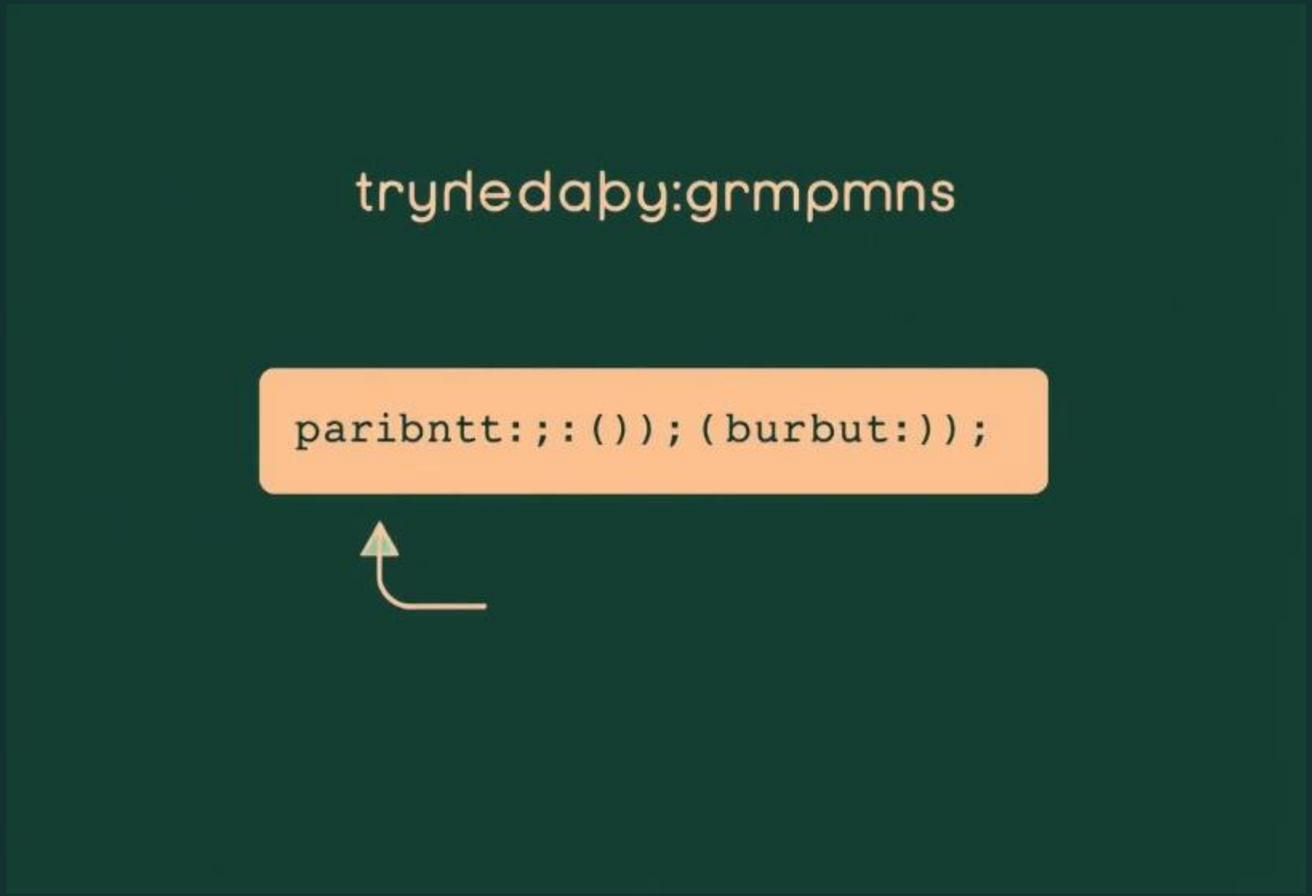
Calling a function executes its code block.

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

int main() {
  int result = sum(5, 3);
  cout << "Sum: " << result;
  return 0;
}
```

Passing Arguments to Functions

Arguments are values passed to a function during its call.

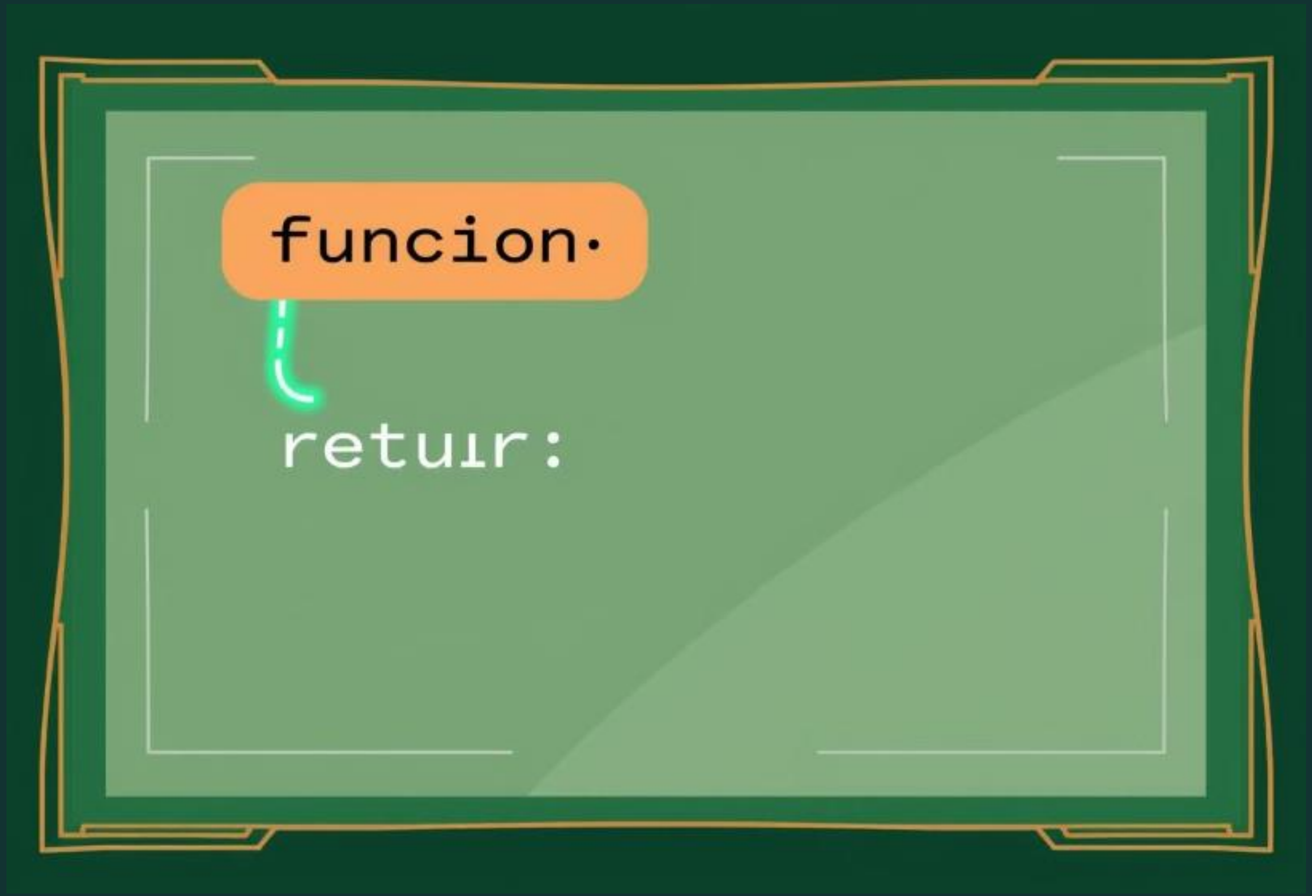
They are used as input for the function's operations.



Returning Values from Functions

Functions can return a value using the **return** statement.

The returned value can be used in the calling code.



Accessing Array Elements

Arrays store collections of elements of the same data type.

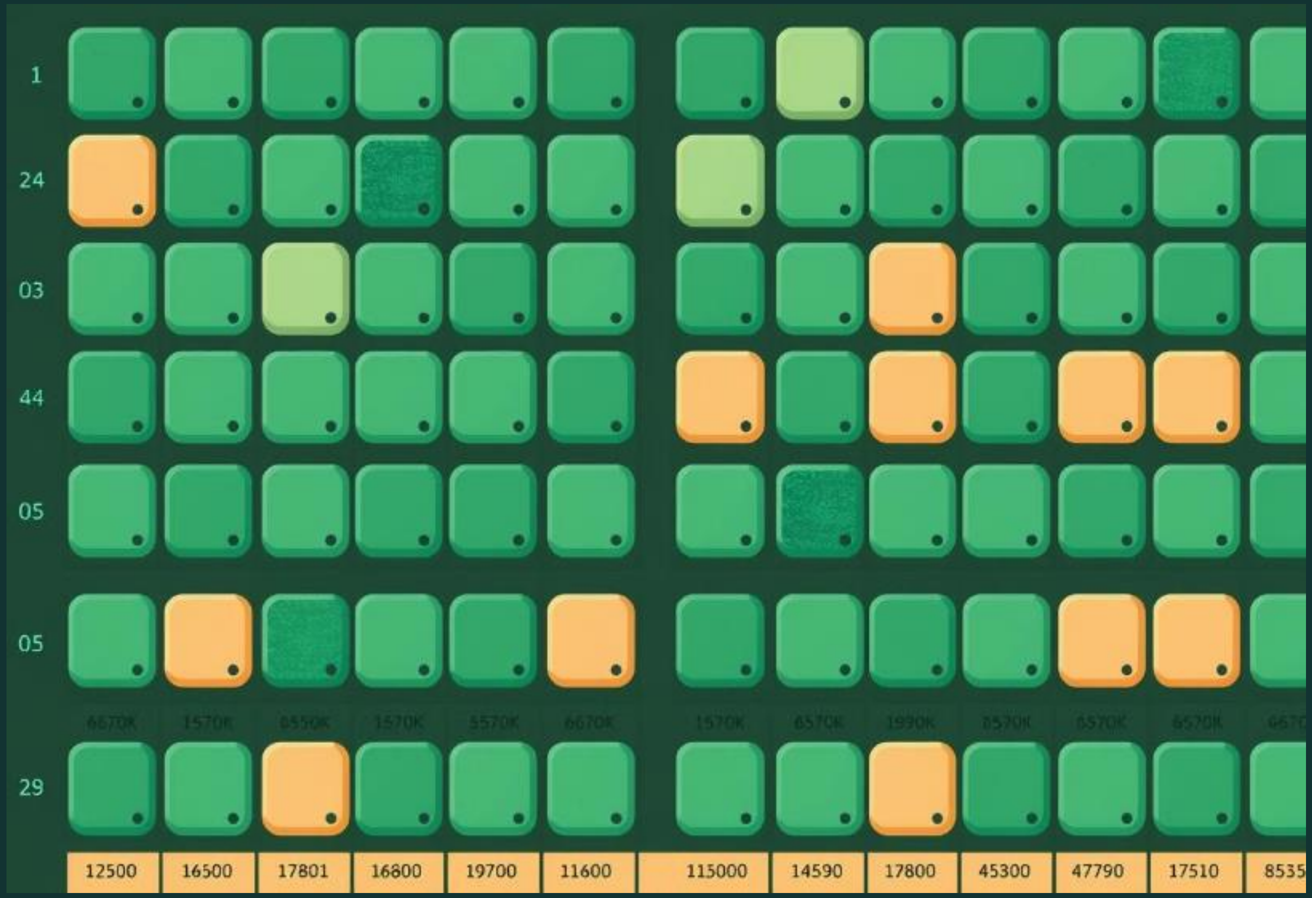
Elements are accessed using their index, starting from 0.

```
int numbers[5] = {10, 20, 30, 40, 50};

cout << "Element at index 2: " <<
numbers[2];
```

2D Arrays: Representing Tabular Data

2D arrays are used for storing tabular data, such as matrices or grids.

They consist of rows and columns, accessed using two indices.



Function Parameters: Passing by Value vs. Reference

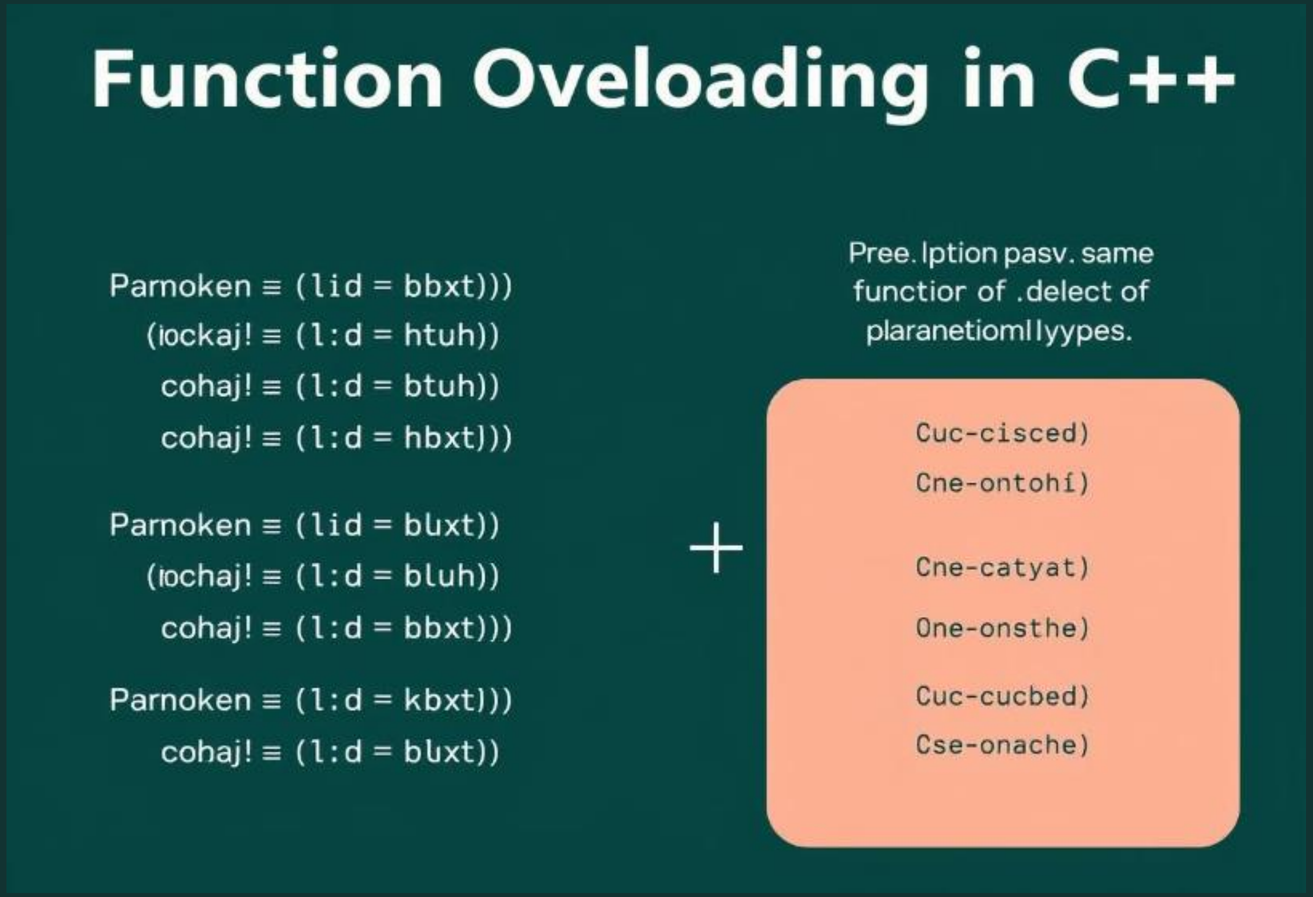
Passing by value creates a copy of the argument, preventing modification of the original.

Passing by reference allows direct modification of the original argument.

Function Overloading: Multiple Functions with the Same Name

Function overloading allows defining multiple functions with the same name.

The compiler selects the correct function based on the argument types.



Defining Overloaded Functions

Overloaded functions have the same name but different parameter lists.

They enhance code reusability by providing different ways to achieve the same result.

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

double sum(double a, double b) {
  return a + b;
}
```

Week 4

Introduction to OOP

Introduction to Object-Oriented Programming

This presentation explores the fundamental concepts of Object-Oriented Programming (OOP), a powerful programming paradigm that provides a structured approach to software development.

Procedural vs. Object-Oriented Programming

Procedural Programming

Focuses on procedures or functions. Data and operations are separate. Data is passed to functions for processing.

Object-Oriented Programming

Emphasizes objects that encapsulate data and behavior. Objects interact with each other through methods.

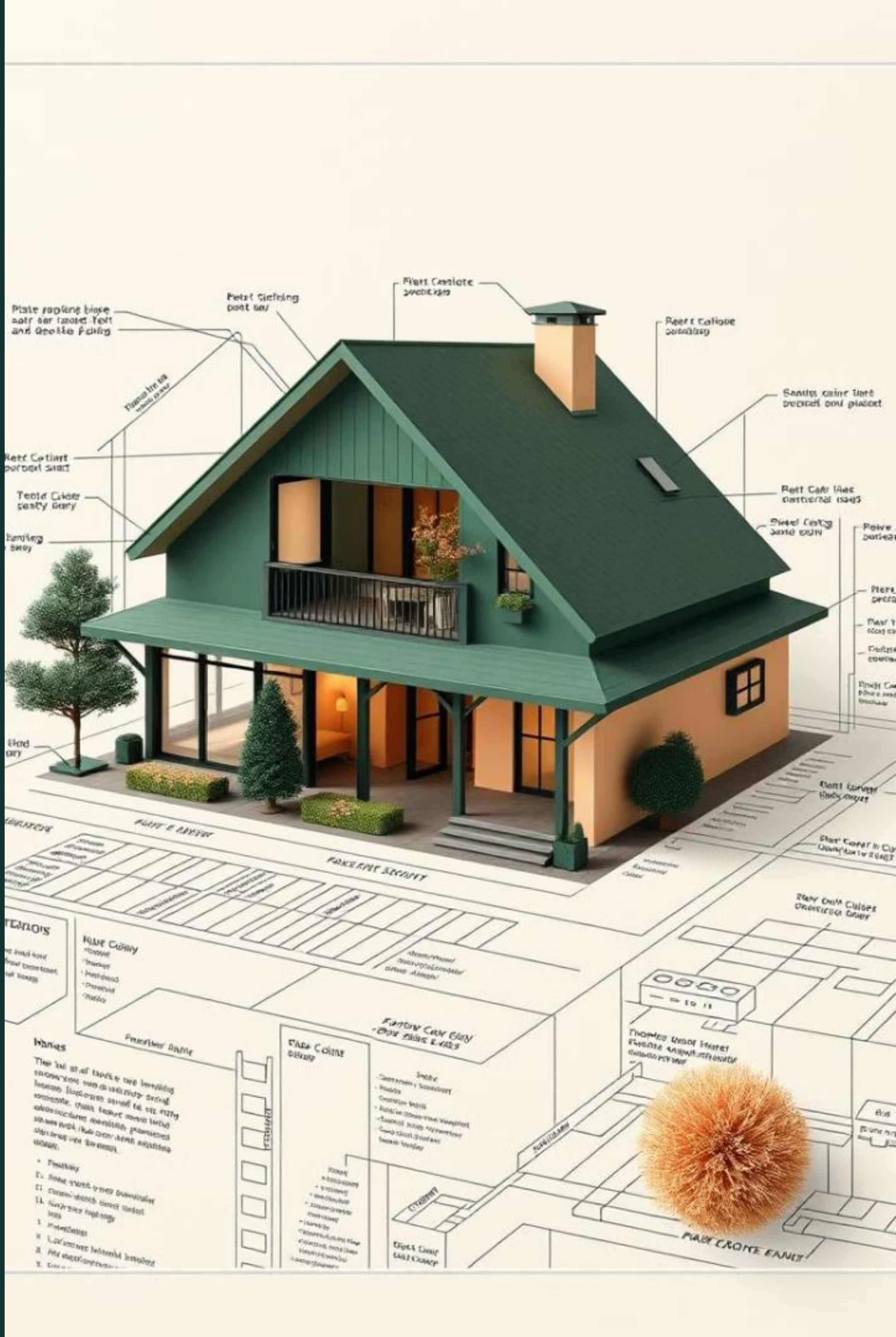
Key Concepts of OOP: Classes and Objects

1. **Class**

   A blueprint or template that defines the structure and behavior of an object.

2. **Object**

   An instance of a class, containing specific data values and methods.



Defining a Class in C++

```
class Dog {
public:
  string name;
  int age;
  void bark() {
    cout << "Woof!" << endl;
  }
};
```

Creating Objects from a Class

```
Dog myDog;
myDog.name = "Buddy";
myDog.age = 3;
myDog.bark();
```

Accessing Class Members

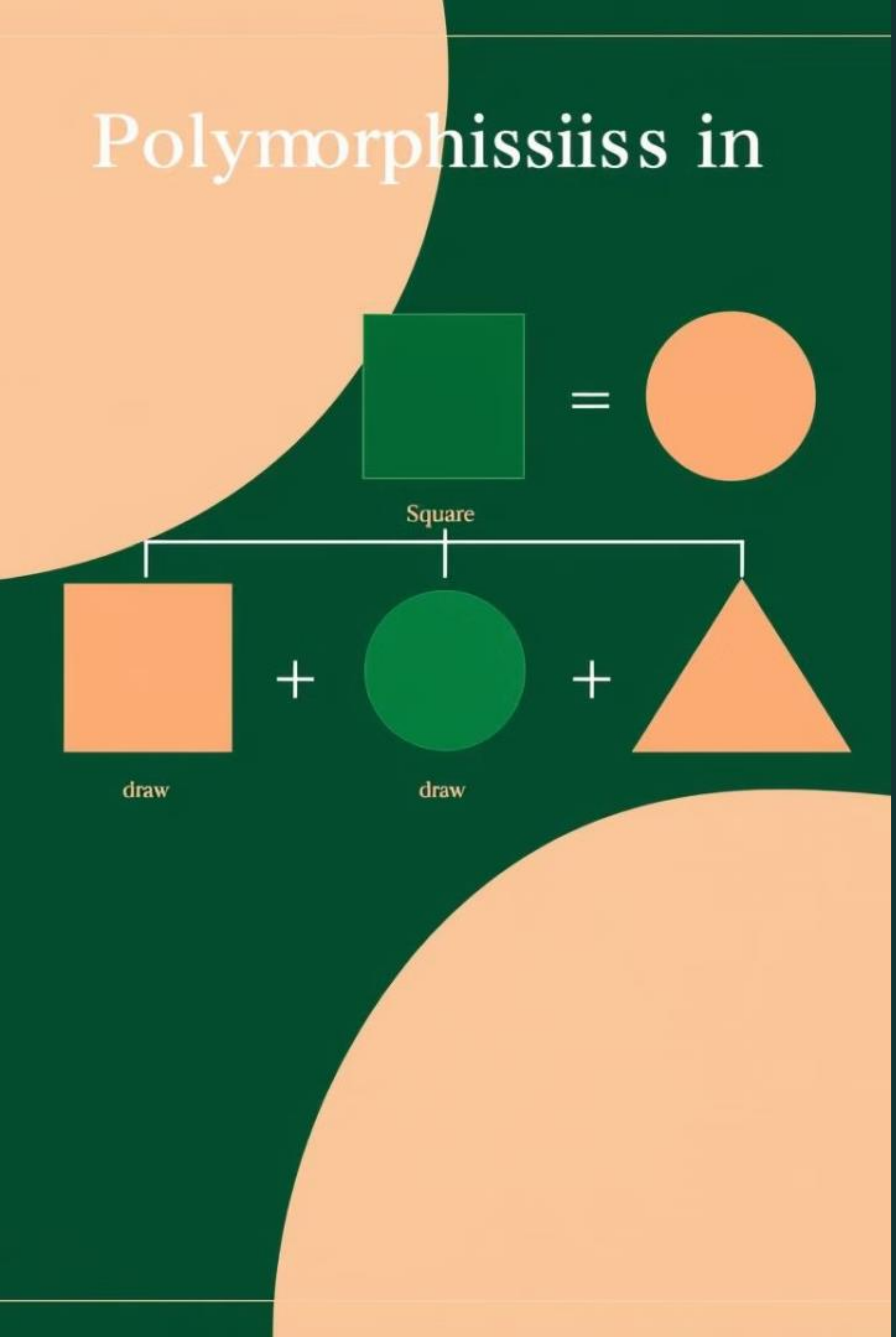
```
class Car {
public:
  string model;
  int year;
  void start() {
    cout << "Engine started." << endl;
  }
};

int main() {
  Car myCar;
  myCar.model = "Ford Mustang";
  myCar.year = 2023;
  myCar.start();
  return 0;
}
```

Constructors and Destructors

```
class Student {
public:
  string name;
  int rollNo;
  Student(string n, int r) {
    name = n;
    rollNo = r;
  }
  ~Student() {
    cout << "Destructor called for " << name << endl;
  }
};
```

Inheritance: Extending Classes

```
class Animal {
public:
  void eat() {
    cout << "Animal eating." << endl;
  }
};

class Dog : public Animal {
public:
  void bark() {
    cout << "Woof!" << endl;
  }
};
```



Polymorphism: Overriding Methods

```
class Shape {
public:
  virtual void draw() = 0;
};

class Circle : public Shape {
public:
  void draw() {
    cout << "Drawing a circle." << endl;
  }
};

class Square : public Shape {
public:
  void draw() {
    cout << "Drawing a square." << endl;
  }
};
```

Conclusion and Key Takeaways

OOP promotes code reusability, modularity, and maintainability. Understanding classes, objects, inheritance, and polymorphism empowers you to build complex and robust software applications.

Week 5

Classes and Objects

Classes and Objects: Constructors, Destructors, Member Functions, and this Pointer

Explore the fundamental building blocks of object-oriented programming in C++, gaining a deep understanding of classes, objects, and their associated concepts.

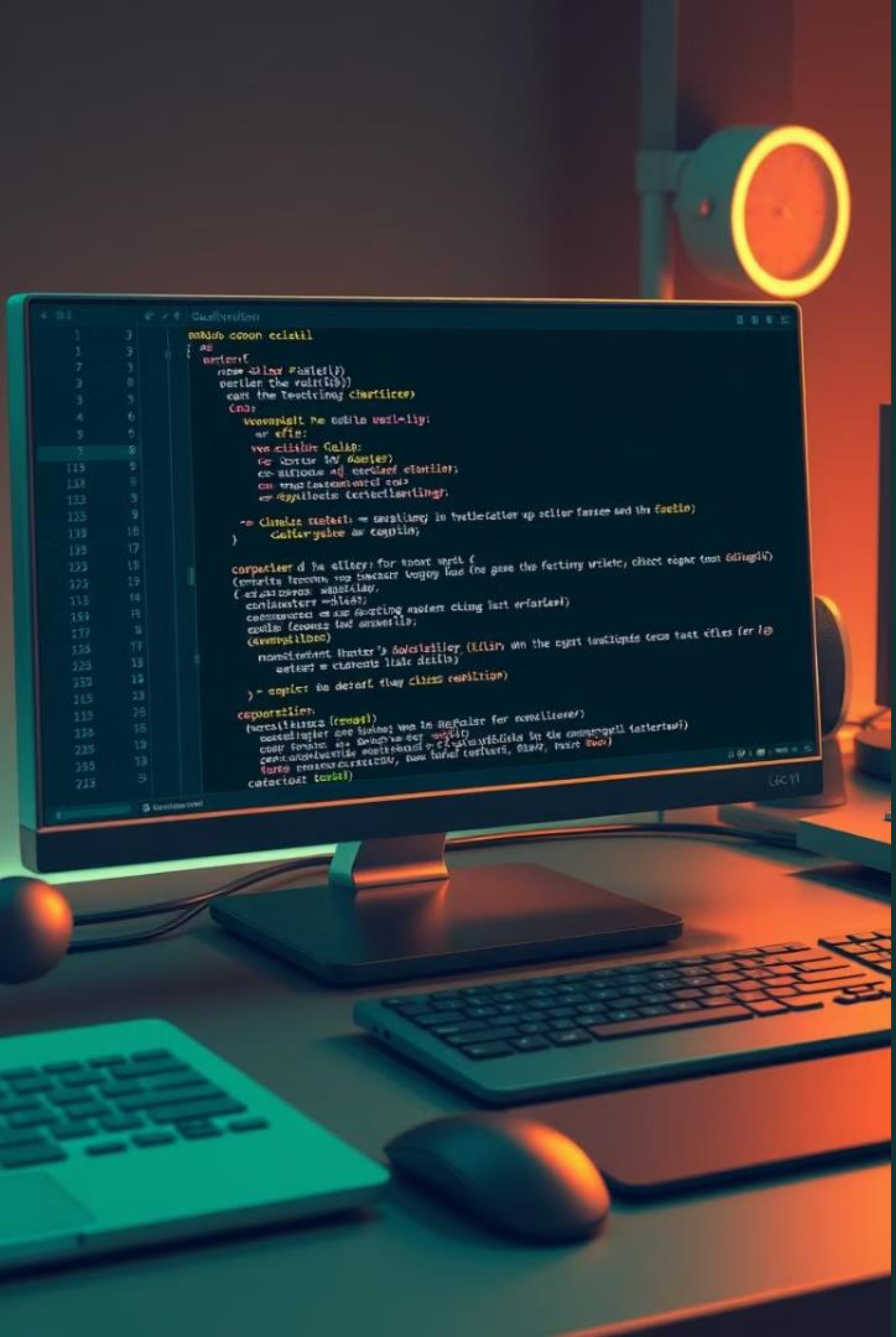
Introduction to Classes and Objects

Classes

Blueprints or templates that define the structure and behavior of objects. They encapsulate data and functions.

Objects

Instances of a class, representing real-world entities. They hold data and can execute the class's functions.



Defining a Class

```
class Car {
  public:
    string brand;
    string model;
    int year;
    void showCar() {
      cout << "Brand: " << brand << endl;
      cout << "Model: " << model << endl;
      cout << "Year: " << year << endl;
    }
};
```

Constructors and Destructors

1

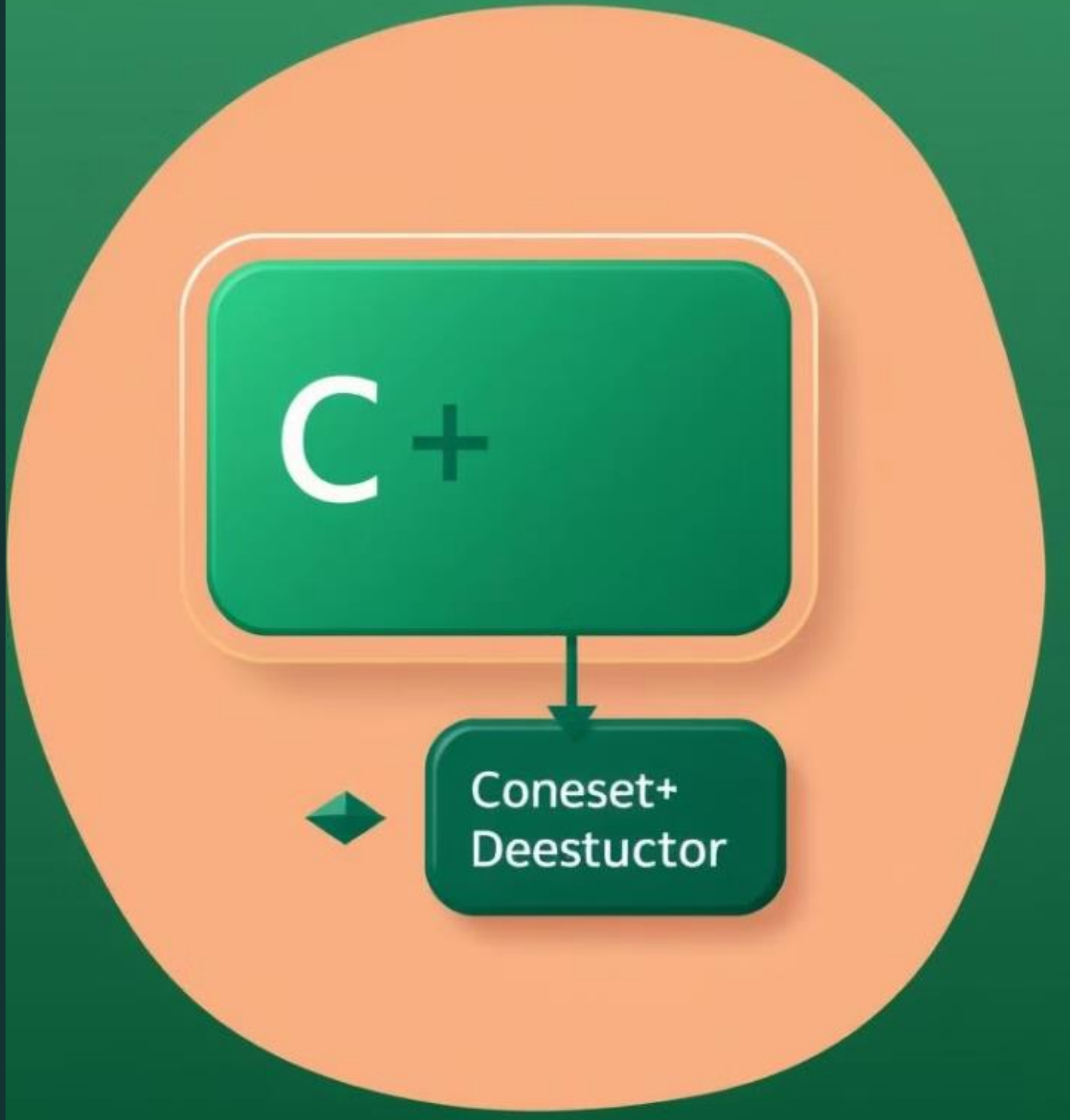
Constructors

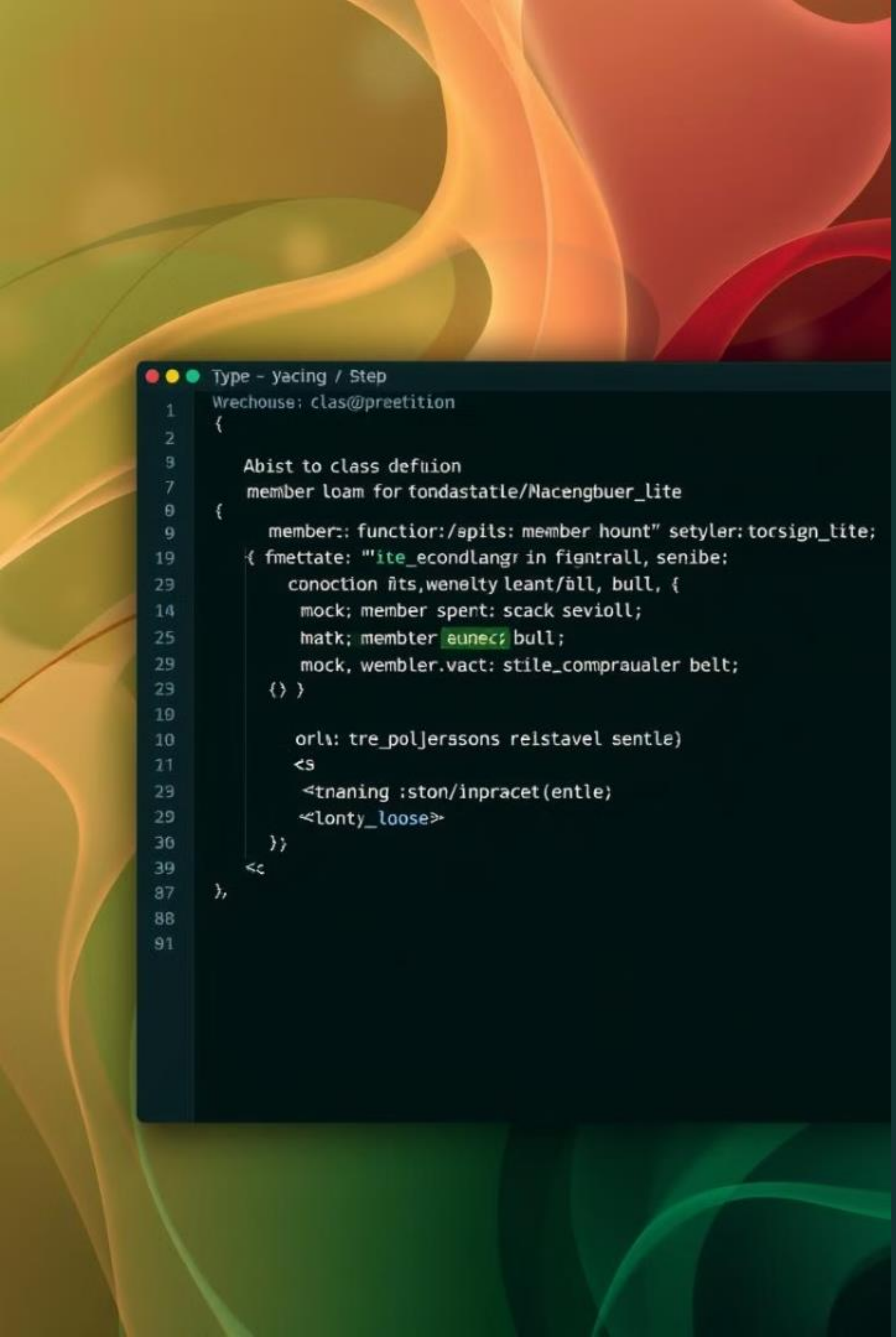
Special member functions that initialize objects when they are created. They have the same name as the class.

2

Destructors

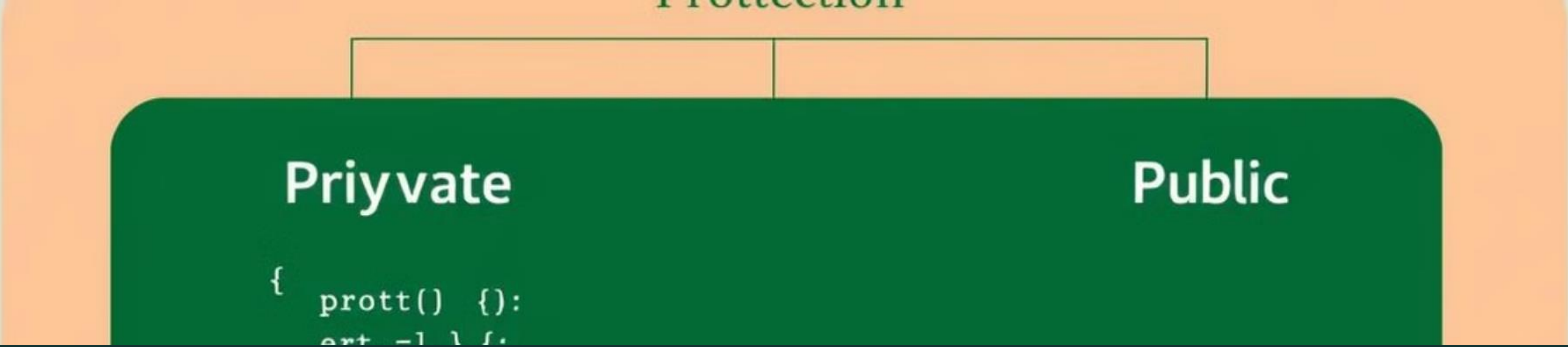
Special member functions that clean up resources when objects are destroyed. They have the same name as the class prefixed with a tilde (~).





Member Functions

```
class Car {
  public:
    Car(string b, string m, int y) { // Constructor
      brand = b;
      model = m;
      year = y;
    }
    void showCar() {  // Member function
      cout << "Brand: " << brand << endl;
      cout << "Model: " << model << endl;
      cout << "Year: " << year << endl;
    }
};
```



Access Specifiers: public, private, protected

public

Members accessible from anywhere, including outside the class.

private

Members accessible only within the class itself.

protected

Members accessible within the class and its derived classes.

The this Pointer

Context

A special pointer available inside member functions that points to the current object.

Purpose

Used to differentiate between member variables and local variables with the same name.

Class Inheritance

1 Base Class

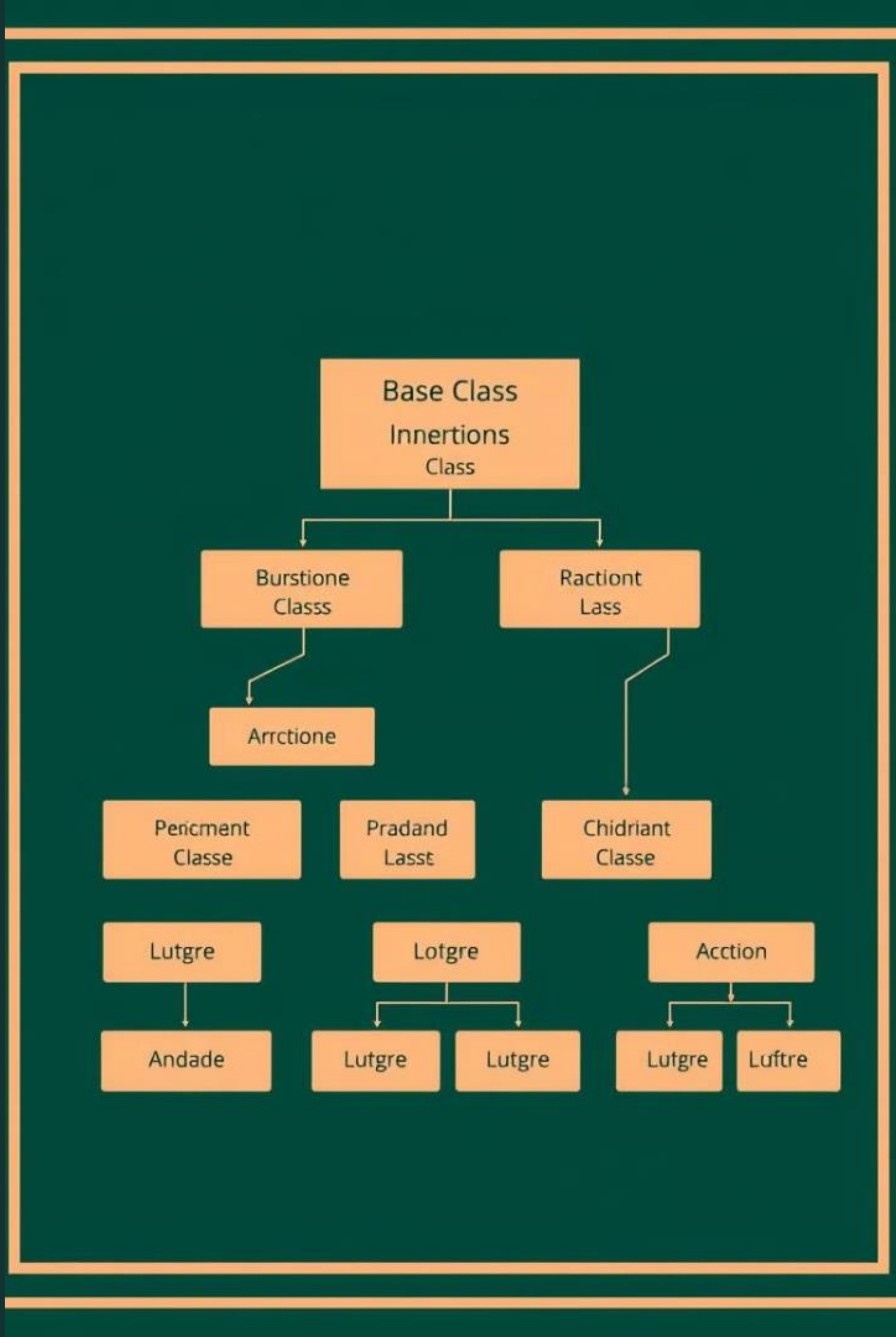
The parent class from which other classes inherit properties and behaviors.

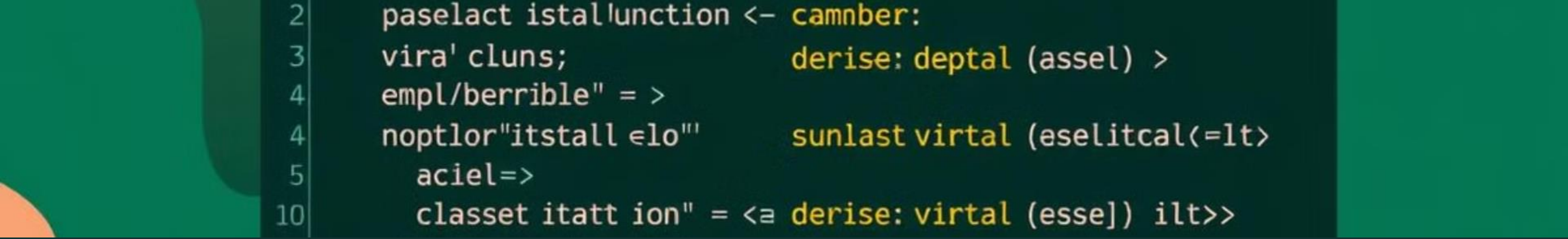
2 Derived Class

The child class that inherits from the base class, extending its functionality.

3 Reusability

Inheritance promotes code reuse by allowing derived classes to use the base class's members.





Polymorphism and Virtual Functions

1

Virtual Functions

Member functions declared with the keyword "virtual" in the base class.

2

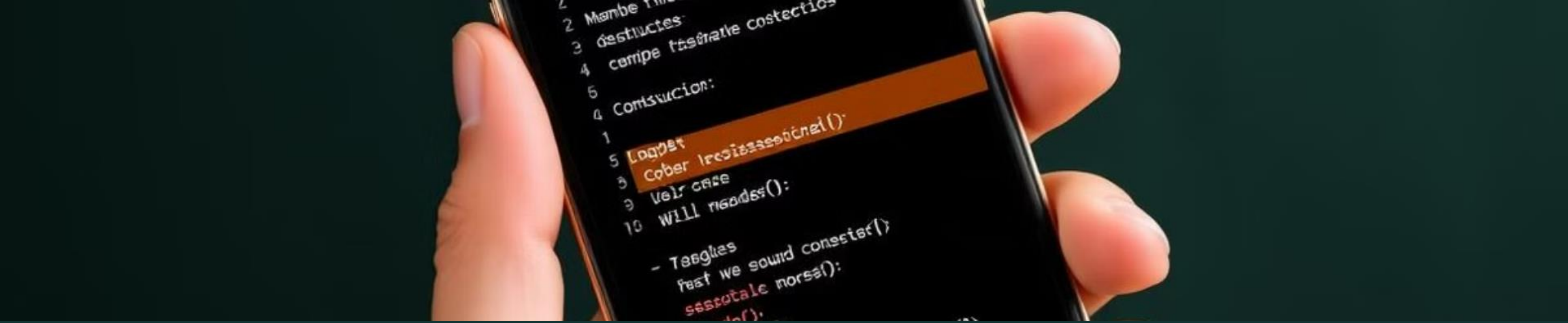
Overriding

Derived classes can provide their own implementations of virtual functions, allowing for dynamic polymorphism.

3

Late Binding

The actual function to be called is determined at runtime, based on the object type.



Code Examples and Live Demonstrations

Let's dive into practical examples and live demonstrations to solidify your understanding of these essential C++ concepts.

Week 6

Inheritance

Inheritance in C++

Inheritance is a powerful C++ concept that enables code reusability and modularity by creating relationships between classes. In this presentation, we will explore inheritance basics, its various types, and key aspects like constructor/destructor chaining and polymorphism.

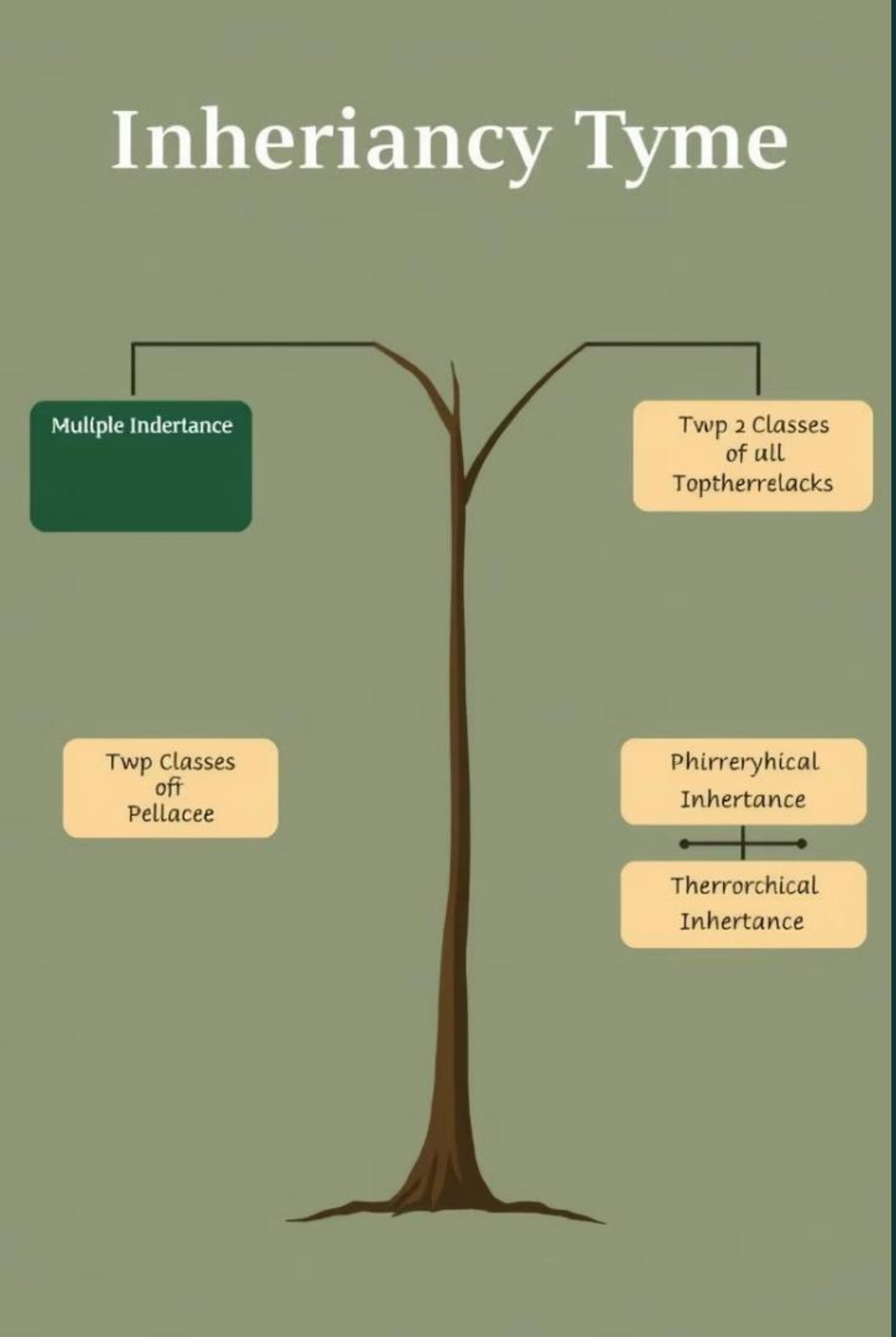
Introduction to Inheritance: Defining Base and Derived Classes

Base Class

The parent class that defines common characteristics and functions. In our example, 'Animal' is the base class.

Derived Class

A class that inherits from a base class, gaining its attributes and functions. 'Dog' is a derived class inheriting from 'Animal'.



Types of Inheritance

Single Inheritance

A single derived class inherits from one base class. For example, 'Dog' inherits from 'Animal'.

Multiple Inheritance

A derived class inherits from multiple base classes. For example, a 'Car' class might inherit from 'Vehicle' and 'Engine' classes.

Hierarchical Inheritance

Multiple derived classes inherit from a single base class. For example, 'Dog', 'Cat', and 'Bird' could all inherit from 'Animal'.

Multilevel Inheritance

A derived class inherits from a base class, and another derived class inherits from the first derived class. For example, a 'SportCar' class could inherit from 'Car', which inherits from 'Vehicle'.

Inheritance and Access Specifiers

Public

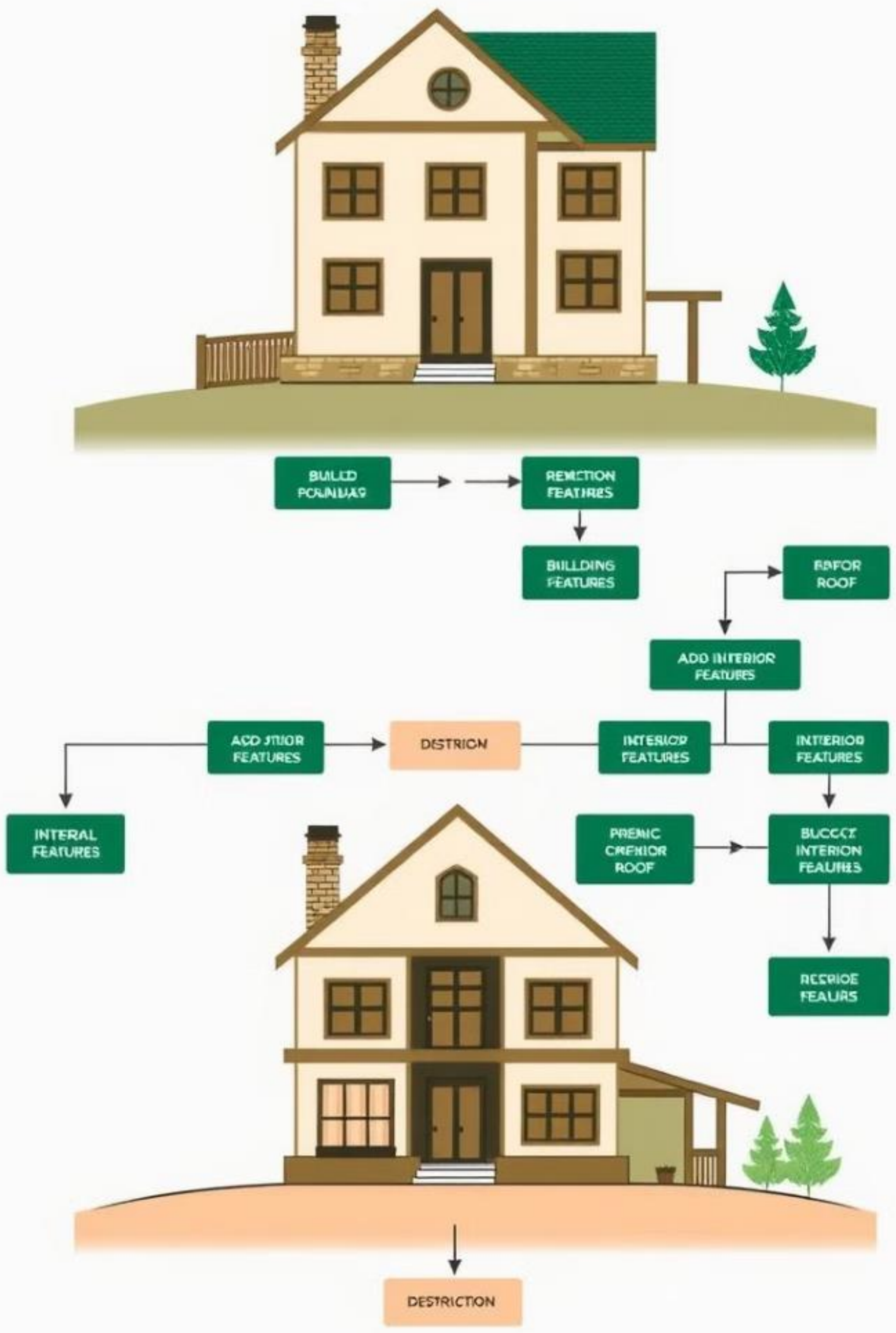
Members declared public in the base class can be accessed directly by derived classes and external code.

Protected

Members declared protected can be accessed by derived classes, but not by external code.

Private

Members declared private are not accessible by derived classes or external code.



Constructors and Destructors in Inheritance

1

Constructor Chaining

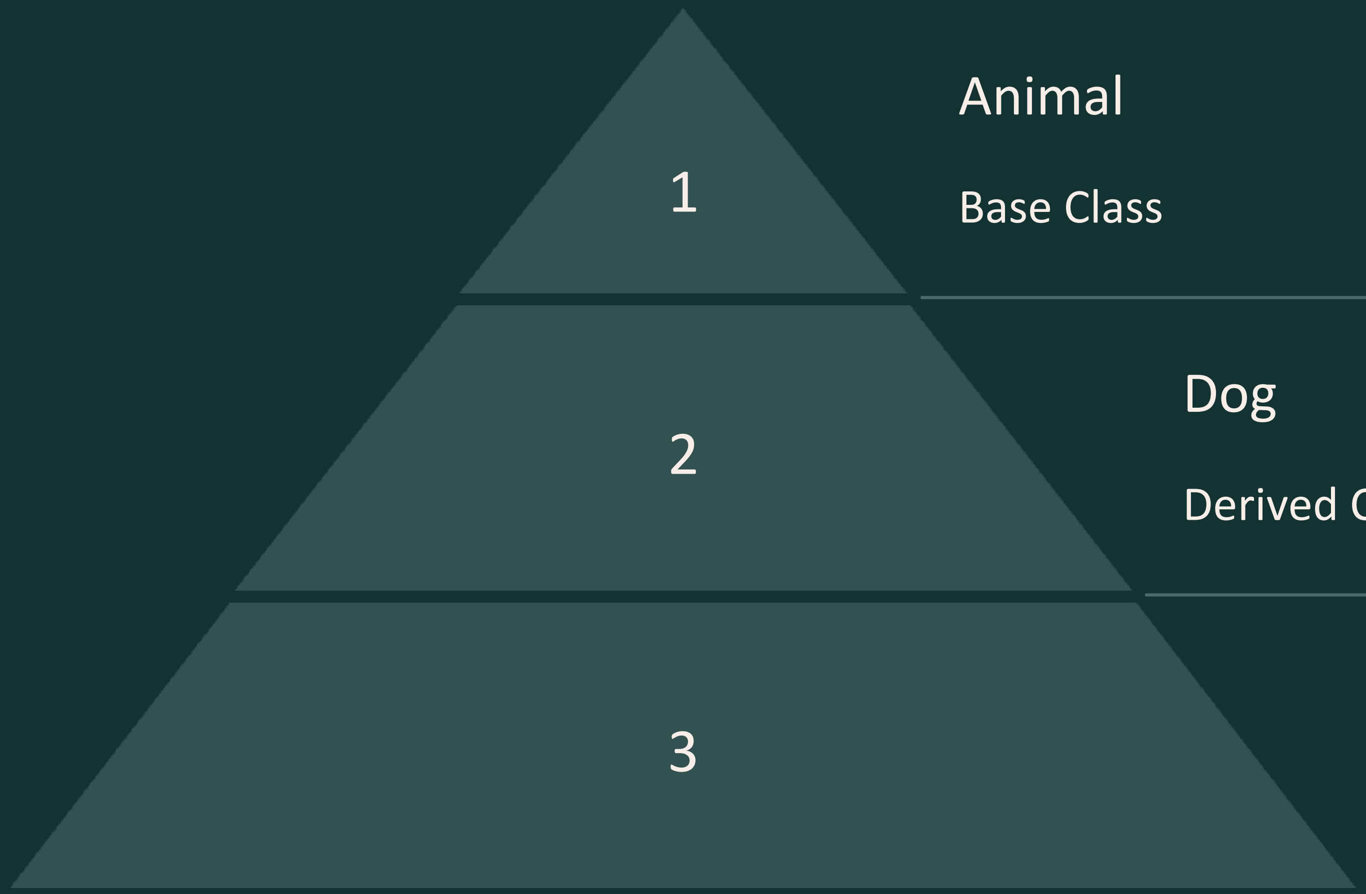
Derived class constructors automatically call the base class constructor.

2

Destructor Execution Order

Destructors are called in the reverse order of constructor execution.

Diagram: Visualizing Inheritance Relationships and Hierarchy



Animal

Base Class

Dog

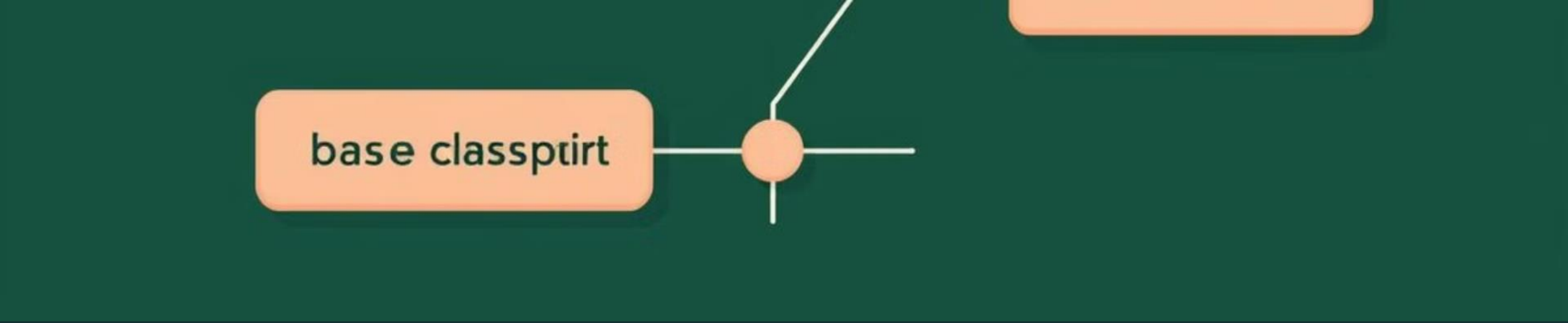
Derived Class

Cat

Derived Class

Inheritance in Code: Examples and Syntax

```
#include <iostream>

class Shape {
public:
  Shape(int sides) : sides(sides) {}
  void printSides() const { std::cout << "Sides: " << sides << std::endl; }

protected:
  int sides;
};

class Triangle : public Shape {
public:
  Triangle() : Shape(3) {}
  void printType() const { std::cout << "Shape: Triangle" << std::endl; }
};

int main() {
  Triangle t;
  t.printType();
  t.printSides(); // Accessing protected member
  return 0;
}
```



Polymorphism and Virtual Functions in Inheritance

Runtime Polymorphism

The ability to call different functions based on the object type at runtime.

Virtual Functions

Functions declared with the 'virtual' keyword in the base class allow for runtime polymorphism.

Advantages and Use Cases of Inheritance in C++

1

Code Reusability

Reduce duplicate code by inheriting from existing classes.

2

Modularity

Create independent and reusable code modules.

3

Extensibility

Easily add new features to existing classes.

Conclusion: Key Takeaways and Further Exploration

Inheritance is a cornerstone of object-oriented programming in C++. It promotes code reusability, modularity, and extensibility, making code more organized and efficient. Dive deeper into inheritance topics like abstract classes, virtual destructors, and multiple inheritance to master its full potential.

Week 7

Polymorphism

Polymorphism in C++

Polymorphism, a core concept in object-oriented programming, empowers code to adapt to different situations and types of objects. This presentation explores the key facets of polymorphism in C++: function overloading, virtual functions, abstract classes, and dynamic method dispatch.

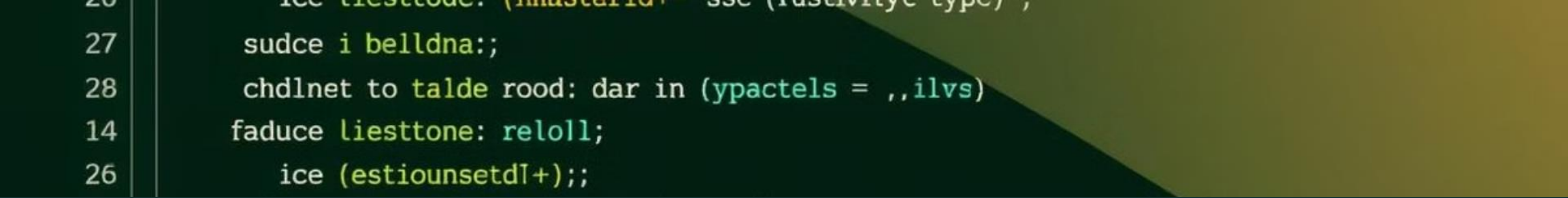
Function Overloading

Same Name, Different Parameters

Function overloading allows defining multiple functions with the same name but distinct parameter lists. This enables using a single function name for diverse functionalities.

Compile-Time Resolution

The C++ compiler determines the appropriate function based on the parameters provided during the function call.



Function Overloading Example

```
#include <iostream>

using namespace std;

int add(int x, int y) {
  return x + y;
}

double add(double x, double y) {
  return x + y;
}

int main() {
  int result1 = add(2, 3); // Calls add(int, int)
  double result2 = add(2.5, 3.5); // Calls add(double, double)
  cout << "result1: " << result1 << endl;
  cout << "result2: " << result2 << endl;
  return 0;
}
```

Virtual Functions

1 **Base Class Function**

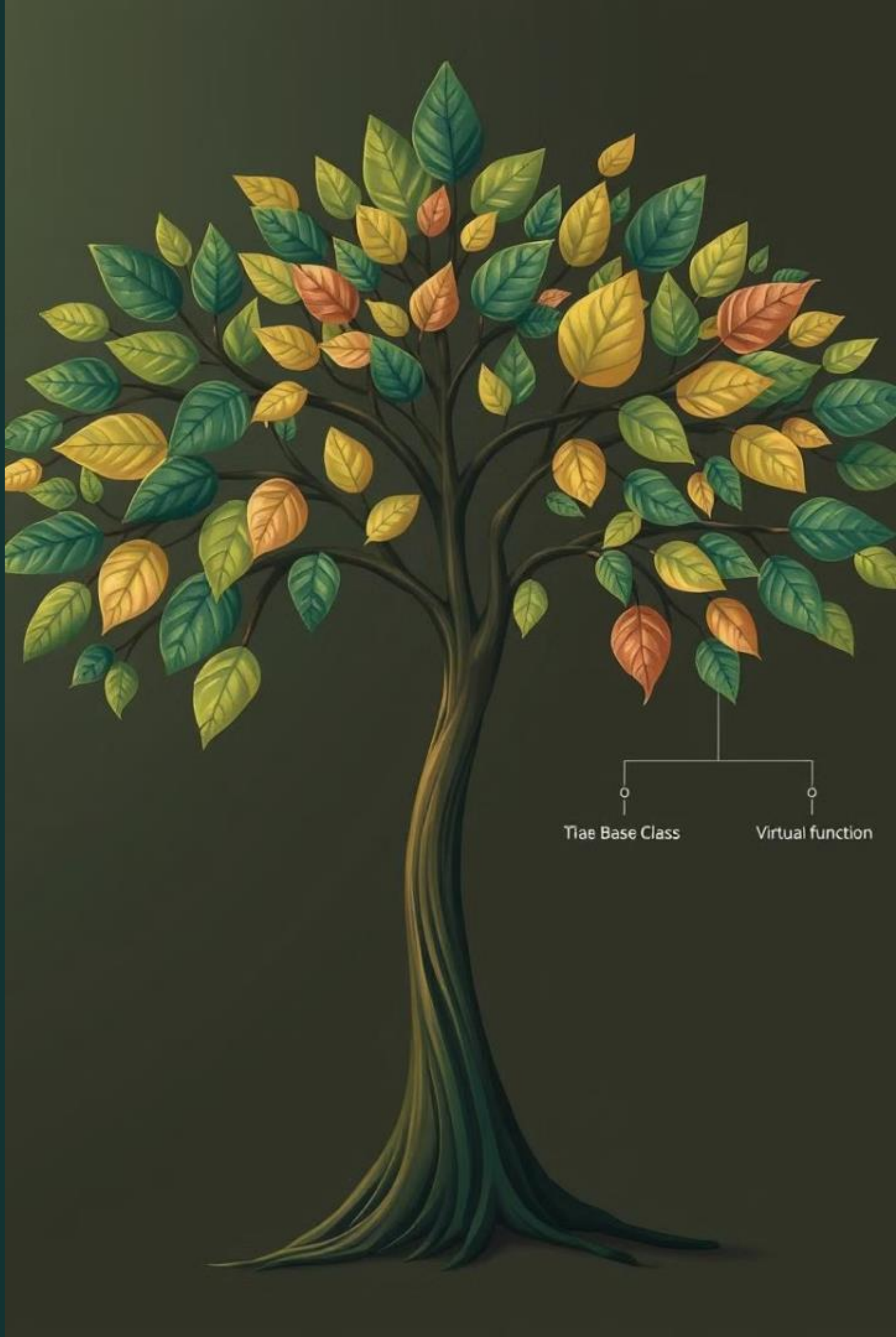
Declaring a function as virtual in the base class enables derived classes to provide their own implementations.

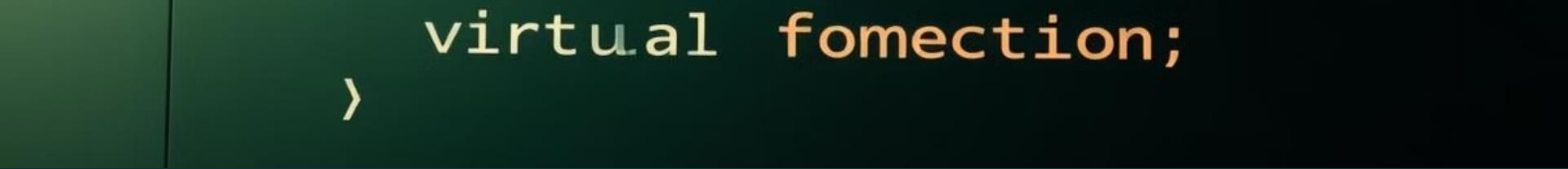
2 **Runtime Polymorphism**

Virtual functions enable runtime polymorphism, where the specific function to execute is determined at runtime.

3 **Overriding Mechanism**

Derived classes can override virtual functions, providing unique behavior for their objects.





Virtual Functions Example

```
#include <iostream>

using namespace std;

class Shape {
public:
  virtual void draw() {
    cout << "Drawing a generic shape" << endl;
  }
};

class Circle : public Shape {
public:
  void draw() {
    cout << "Drawing a circle" << endl;
  }
};

int main() {
  Shape* shape1 = new Shape();
  Shape* shape2 = new Circle();
  shape1->draw(); // Calls Shape::draw()
  shape2->draw(); // Calls Circle::draw()
  return 0;
}
```

Abstract Classes

1 Uninstantiable Base Class

Abstract classes cannot be instantiated, acting as blueprints for derived classes.

2 Pure Virtual Functions

Abstract classes contain pure virtual functions, which must be implemented by derived classes.

3 Encapsulation of Behavior

Abstract classes enforce a common interface and ensure derived classes implement specific behaviors.

Abstract Classes Example

```cpp
#include <iostream>

using namespace std;

class Animal {
public:
  virtual void makeSound() = 0; // Pure virtual function
};

class Dog : public Animal {
public:
  void makeSound() {
    cout << "Woof!" << endl;
  }
};

int main() {
  // Animal animal; // Error: Cannot instantiate abstract class
  Dog dog;
  dog.makeSound();
  return 0;
}
```

Dynamic Method Dispatch

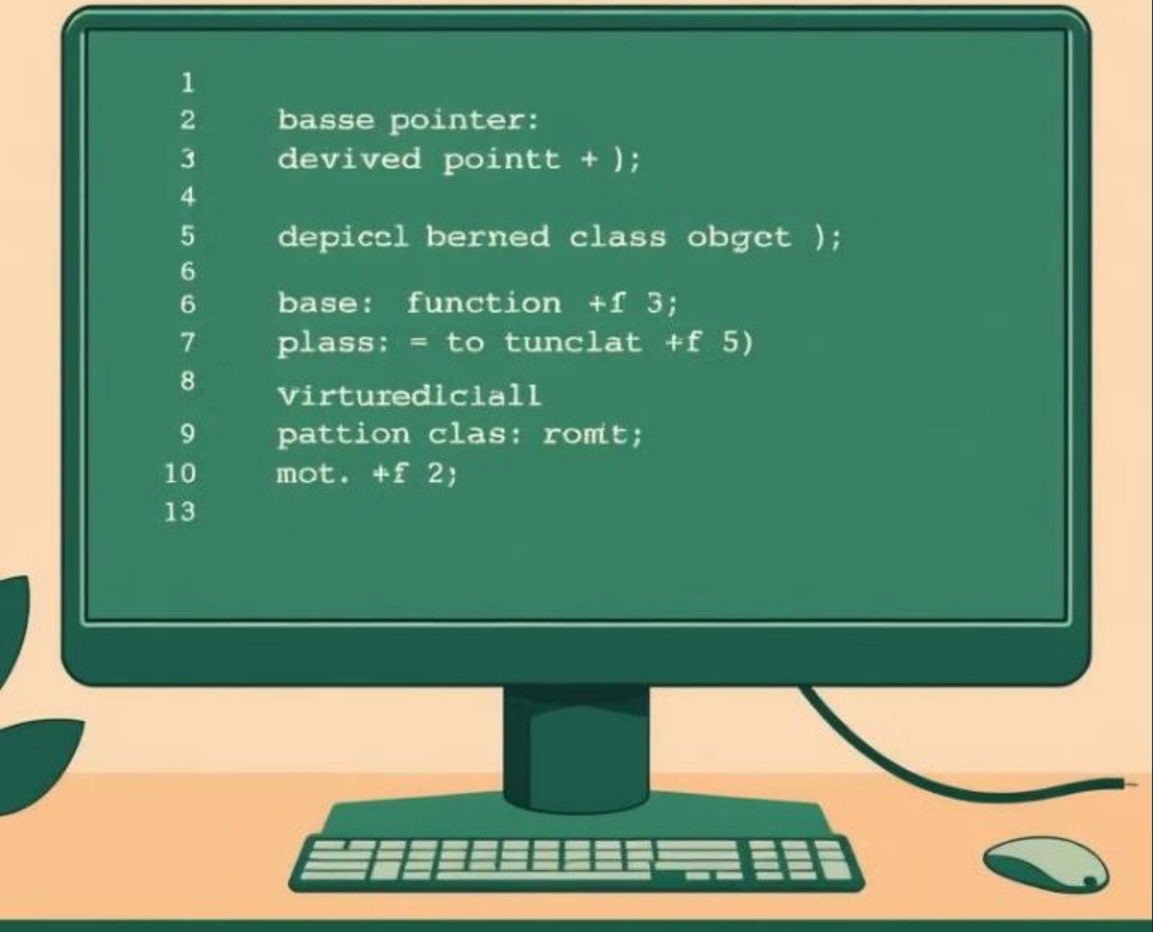
1. ### Runtime Resolution

2. ### Virtual Function Table (vtable)

   Dynamic method dispatch utilizes a virtual function table (vtable) to determine the correct function implementation based on the object's type at runtime.

3. ### Polymorphic Behavior

   This process enables polymorphic behavior, where the same code can interact with objects of different derived classes in a consistent way.



Dynamic Method Dispatch Example

```
#include <iostream>

using namespace std;

class Shape {
public:
  virtual void draw() {
    cout << "Drawing a generic shape" << endl;
  }
};

class Circle : public Shape {
public:
  void draw() {
    cout << "Drawing a circle" << endl;
  }
};

class Square : public Shape {
public:
  void draw() {
    cout << "Drawing a square" << endl;
  }
};

int main() {
  Shape* shapes[2];
  shapes[0] = new Circle();
  shapes[1] = new Square();
  for (int i = 0; i < 2; i++) {
    shapes[i]->draw();
  }
  return 0;
}
```

Polymorphism: A Visual Summary

Function Overloading

Multiple functions with same name, different parameters, resolved at compile-time.

Virtual Functions

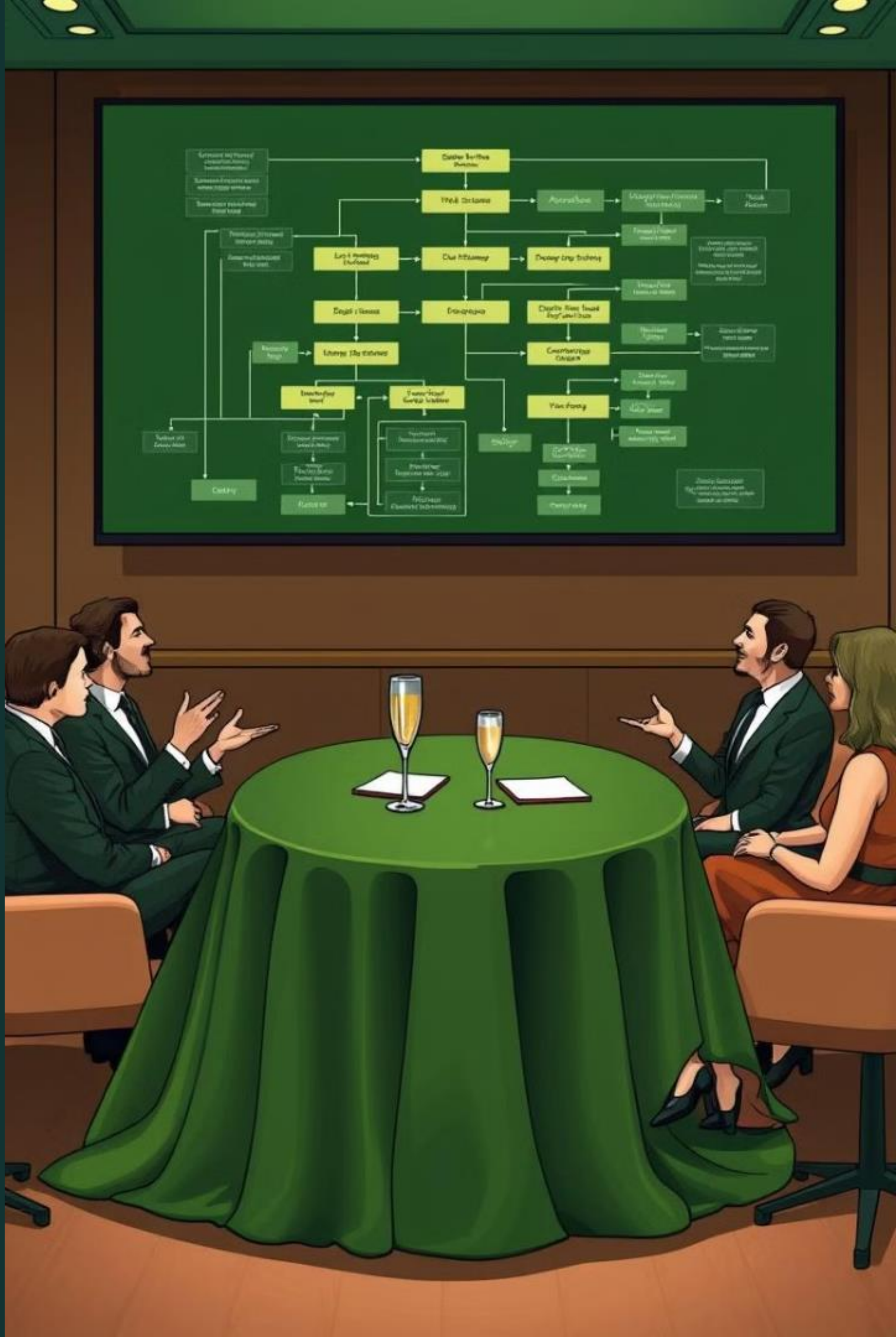
Base class functions that can be overridden by derived classes, resolved at runtime.

Abstract Classes

Uninstantiable base classes with pure virtual functions, enforcing common interfaces for derived classes.

Dynamic Method Dispatch

Resolving the appropriate function implementation at runtime, based on the object's type, using vtables.



Week 8

Encapsulation

Encapsulation: Grouping Data and Access Control

This presentation will explore encapsulation, a fundamental concept in object-oriented programming (OOP) that enhances code organization, security, and maintainability. It involves grouping data and the functions that operate on that data within a single unit, a class, and controlling access to this data.

Introduction to Encapsulation

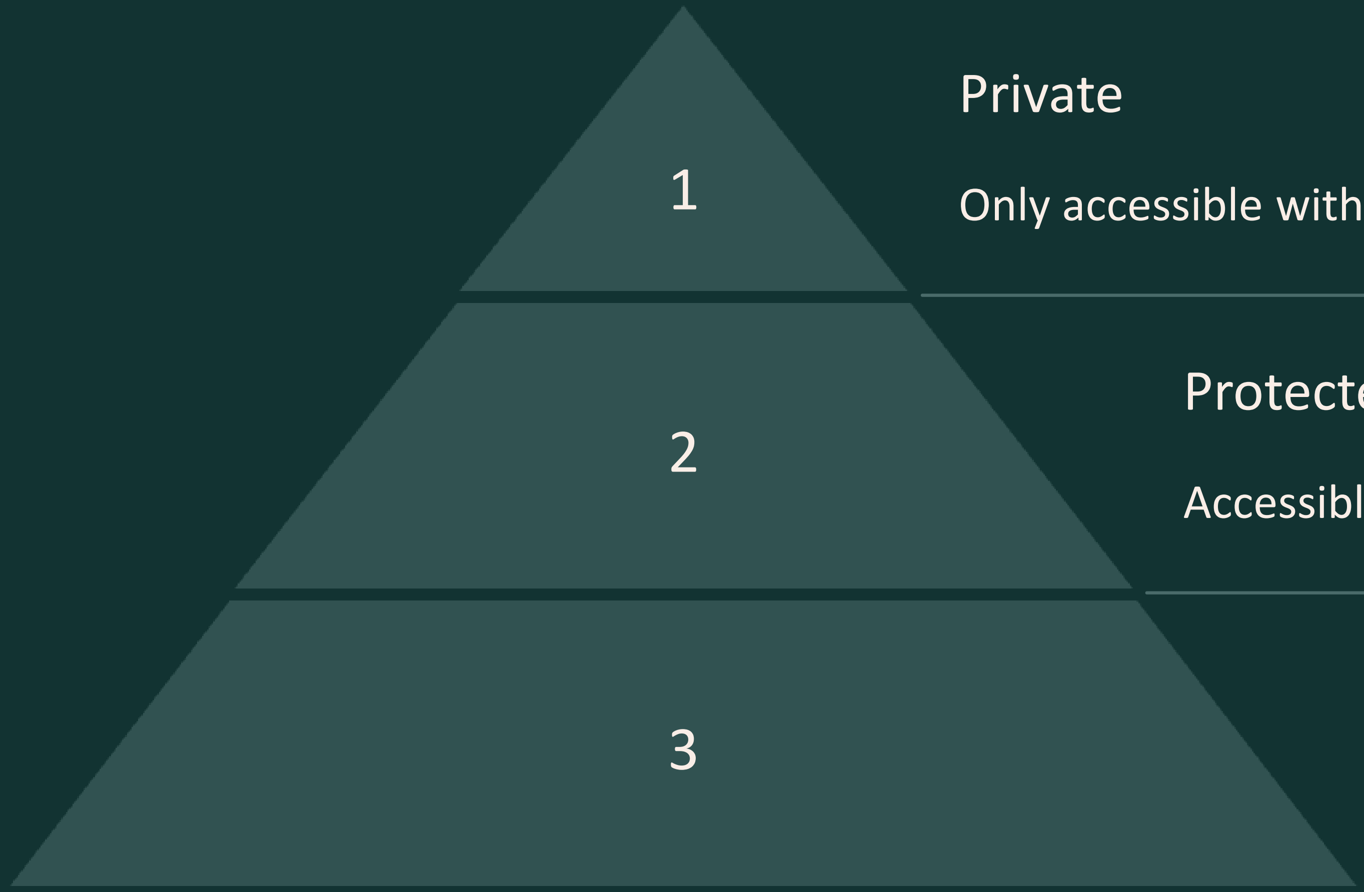
Data Hiding

Encapsulation helps protect data from unauthorized access and modification by making it private, ensuring data integrity and consistency.

Code Organization

It promotes modularity and code reusability by grouping related data and functions together, improving code structure and maintainability.

Data Encapsulation: Private, Protected, and Public



Private

Only accessible within the class itself.

Protected

Accessible within the class and its derived classes.

Public

Accessible from anywhere, even outside the class.

Accessing Class Members: Public vs. Private

Private Members

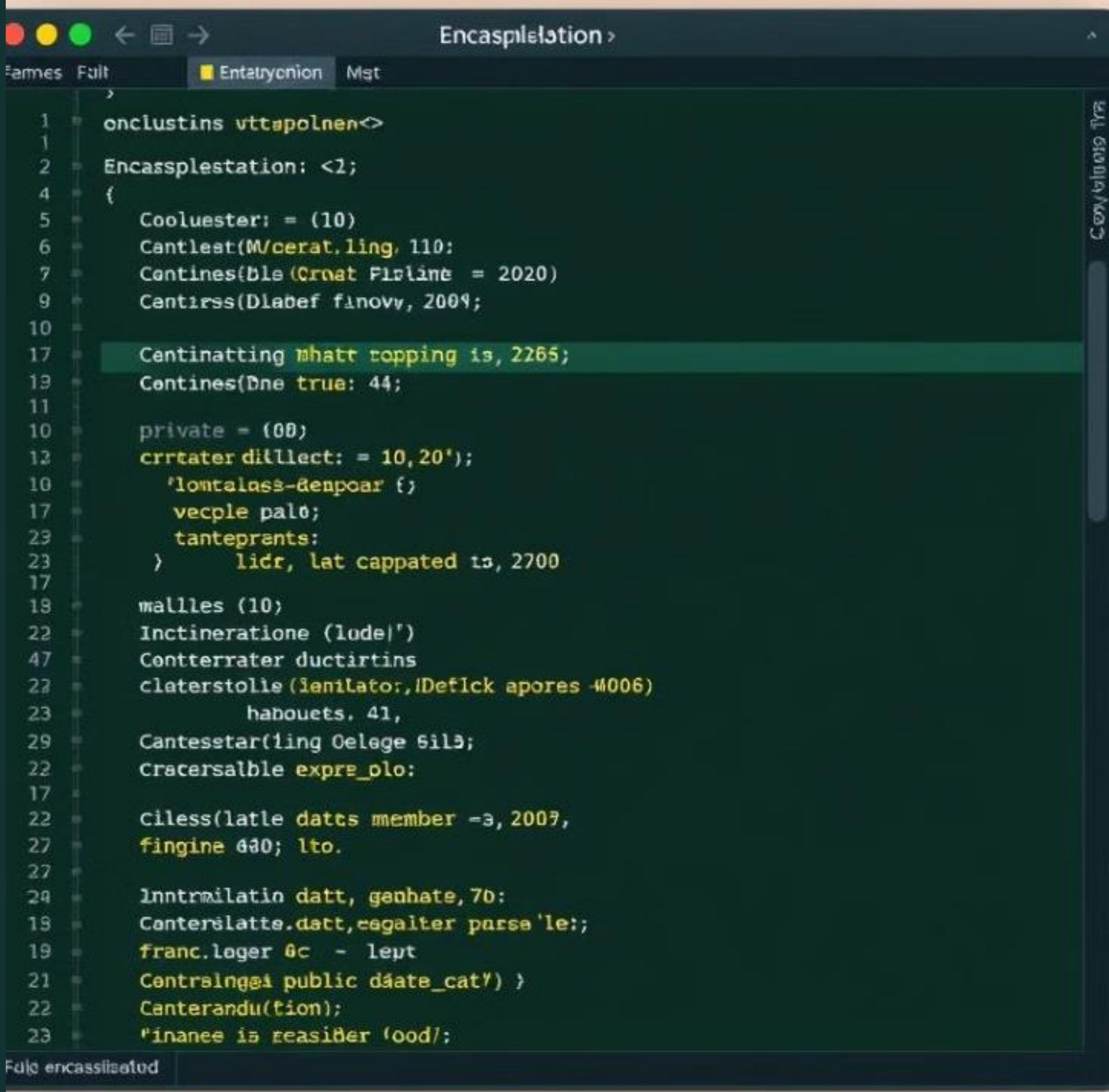
Cannot be directly accessed from outside the class.

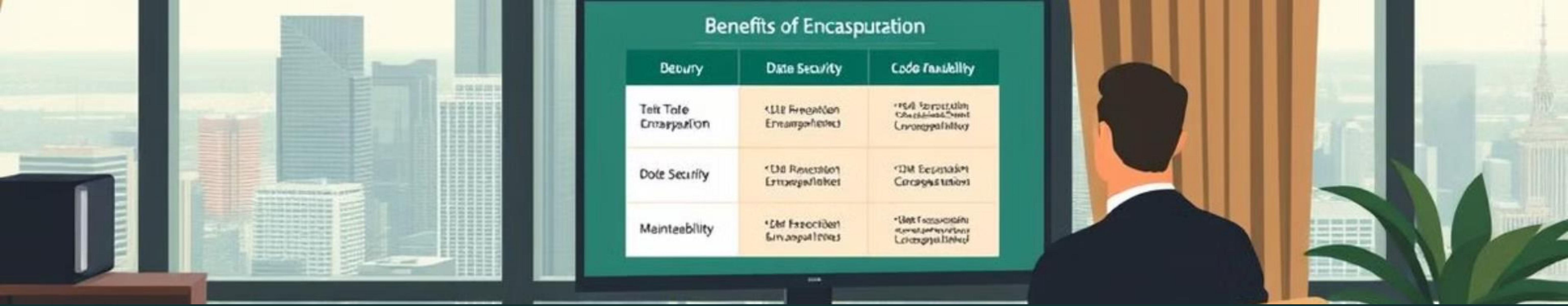
Public Members

Can be accessed directly from outside the class. These are typically getter and setter functions to control access to private data.

Demonstration: Encapsulation in C++

```
#include

class Employee {
private:
    int empId;
    std::string name;

public:
    void setEmpId(int id) { empId = id; }
    int getEmpId() { return empId; }
    void setName(std::string n) { name = n; }
    std::string getName() { return name; }
};

int main() {
    Employee employee;
    employee.setEmpId(123);
    employee.setName("Alice");

    std::cout << "Employee ID: " << employee.getEmpId() << std::endl;
    std::cout << "Employee Name: " << employee.getName() << std::endl;
    return 0;
}
```





Benefits of Encapsulation

1 **Data Protection**

Shields internal data from unauthorized access and modification, ensuring data integrity.

2 **Modularity**

Encapsulation promotes modularity, making code easier to understand, maintain, and debug.

3 **Code Reusability**

Encapsulated classes can be reused across different projects, reducing code duplication.

4 **Flexibility**

Encapsulation allows for changes to internal implementation without affecting external code.

Encapsulation and Information Hiding

1

Data Hiding

Key concept behind encapsulation. Prevents direct access to internal data members, ensuring data integrity.

2

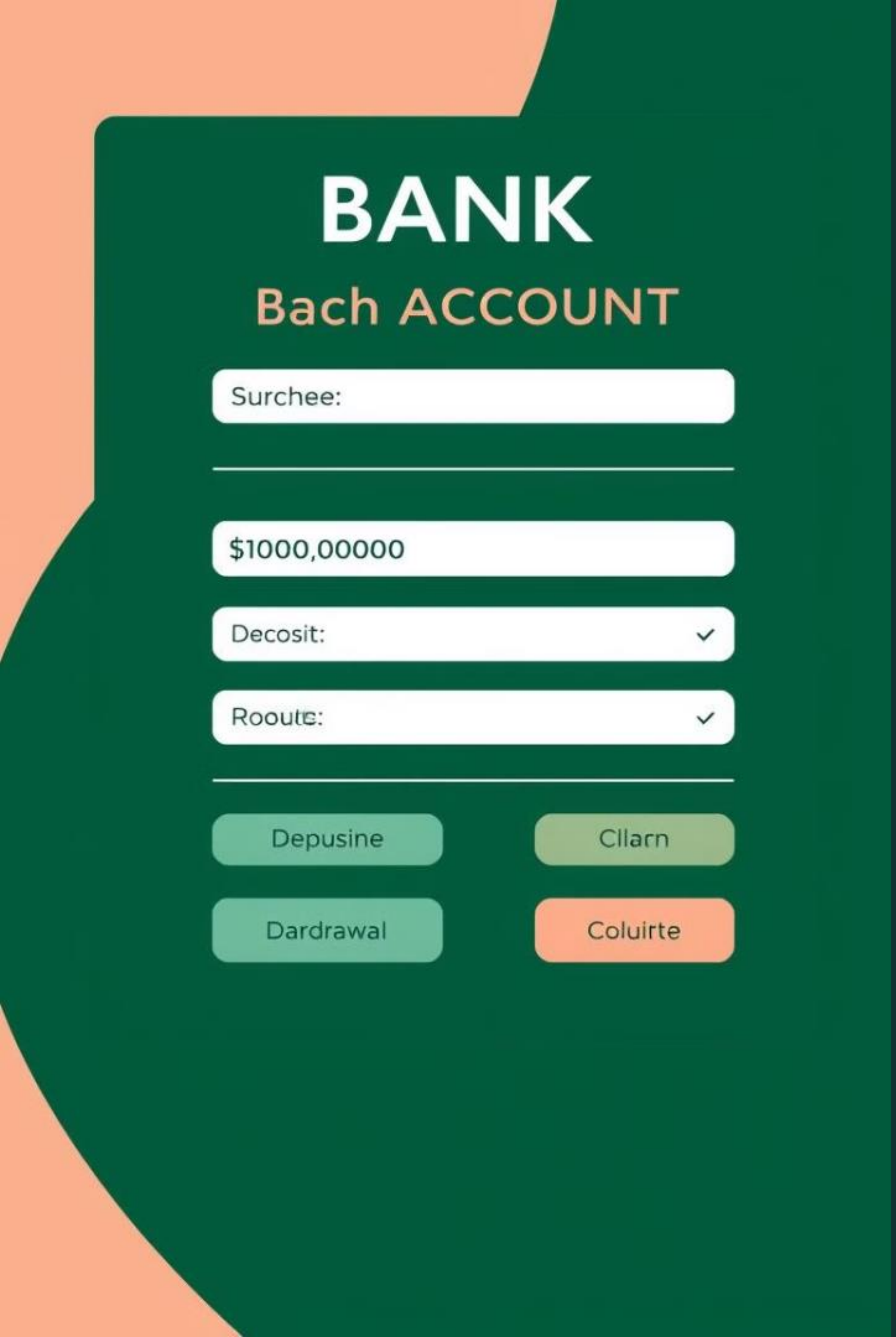
Controlled Access

Provides controlled access to data through publicly exposed methods (getter and setter functions).

3

Maintainability

Simplifies code maintenance by allowing changes to internal implementation without impacting external code.



Practical Example: Encapsulating a Bank Account

Data Members

account number, balance, etc.

Public Methods

deposit(), withdraw(), getBalance()

Information Hiding

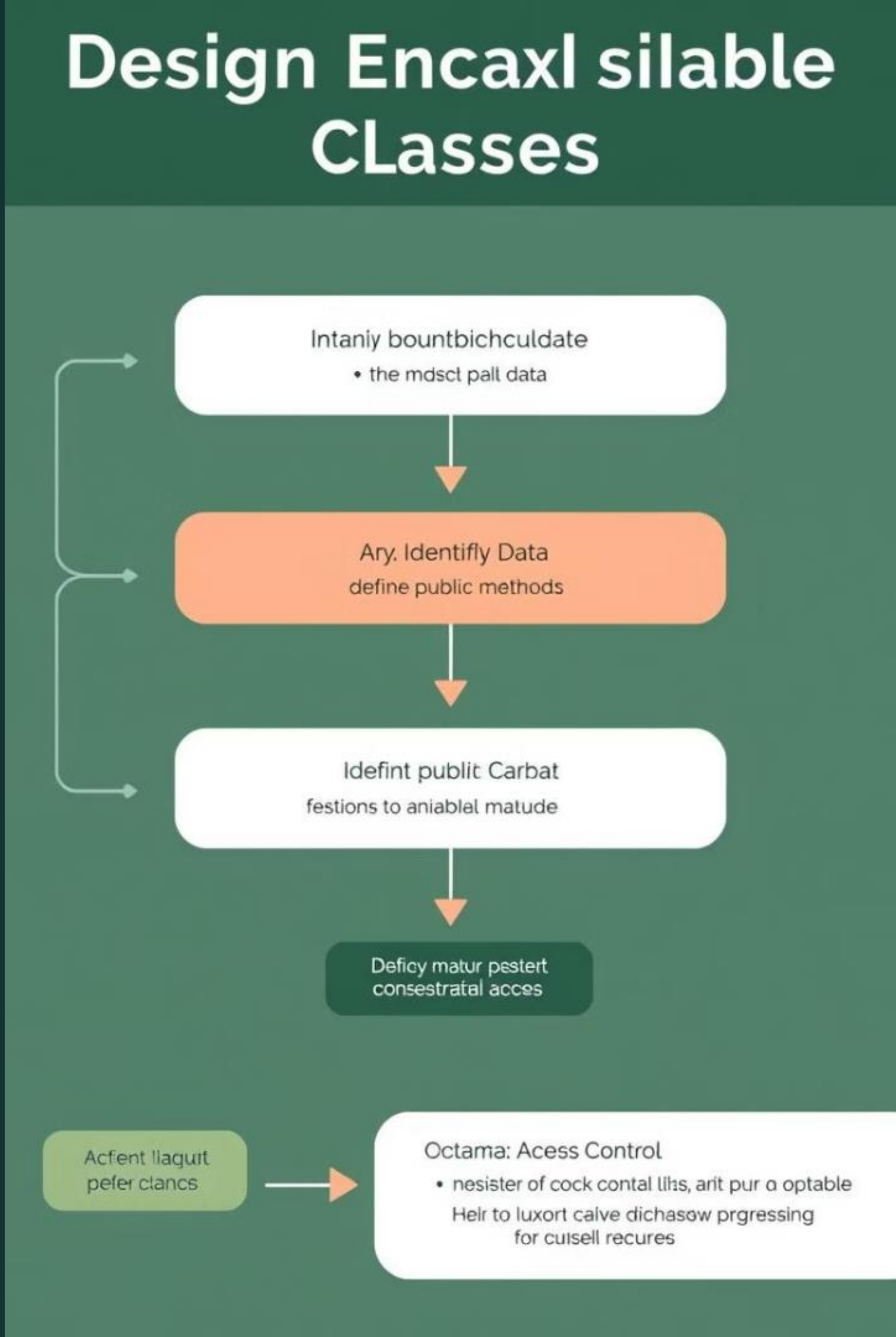
Internal data members are private, accessed only through public methods.

Designing Encapsulated Classes

1. Define the data members, representing the state of the object.

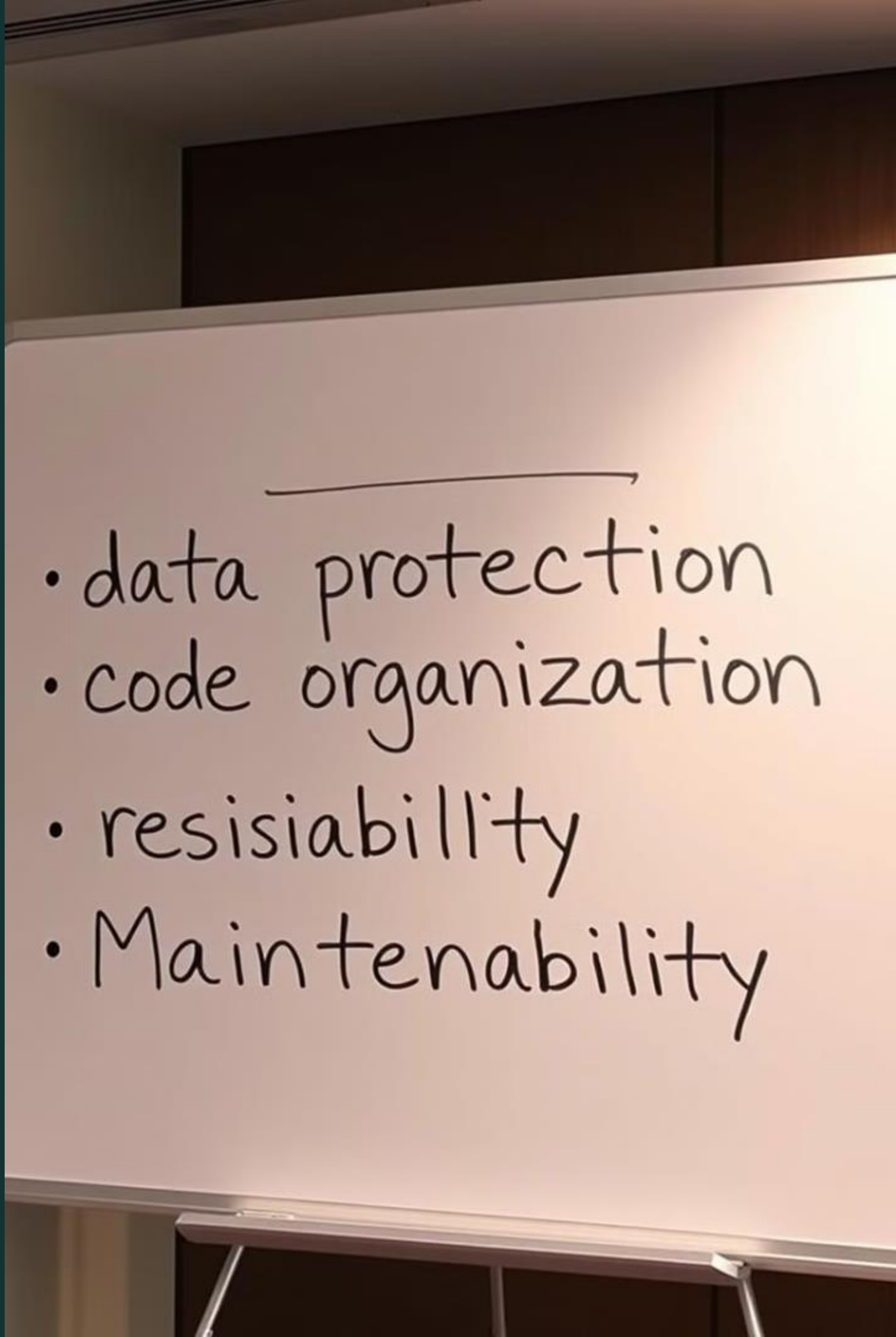
2. Implement the public methods to control access to data members and provide functionality.

3. Consider access modifiers (private, protected, public) to determine what parts of the class are accessible from outside.



Conclusion and Key Takeaways

Encapsulation is a fundamental OOP concept that promotes data protection, code organization, reusability, and maintainability. By carefully defining data and methods, and controlling access through public interfaces, developers can build robust and maintainable software systems. Understanding and applying encapsulation principles is essential for building reliable and scalable software.



Week 9

Abstraction

Abstraction in C++

This presentation will cover the concept of abstraction in C++, exploring abstract classes and interfaces.

What is Abstraction?

Hiding Complexity

Abstraction simplifies complex systems by hiding implementation details. You only interact with the essential features.

Focus on Behavior

It emphasizes what an object does, rather than how it does it. It's like using a remote control without knowing how the TV works.

Importance of Abstraction

Code Reusability

Abstraction allows you to create reusable components, reducing code duplication and improving maintainability.

Flexibility

Abstraction makes code more adaptable to changes. You can easily modify implementation details without affecting the overall behavior.

Maintainability

By separating concerns, abstraction makes it easier to understand, debug, and modify complex systems.

Abstract Classes in C++

1. **Blueprint**

   An abstract class acts as a blueprint for derived classes.

2. **Incomplete Implementation**

   It defines the structure and behavior but doesn't provide all the details.

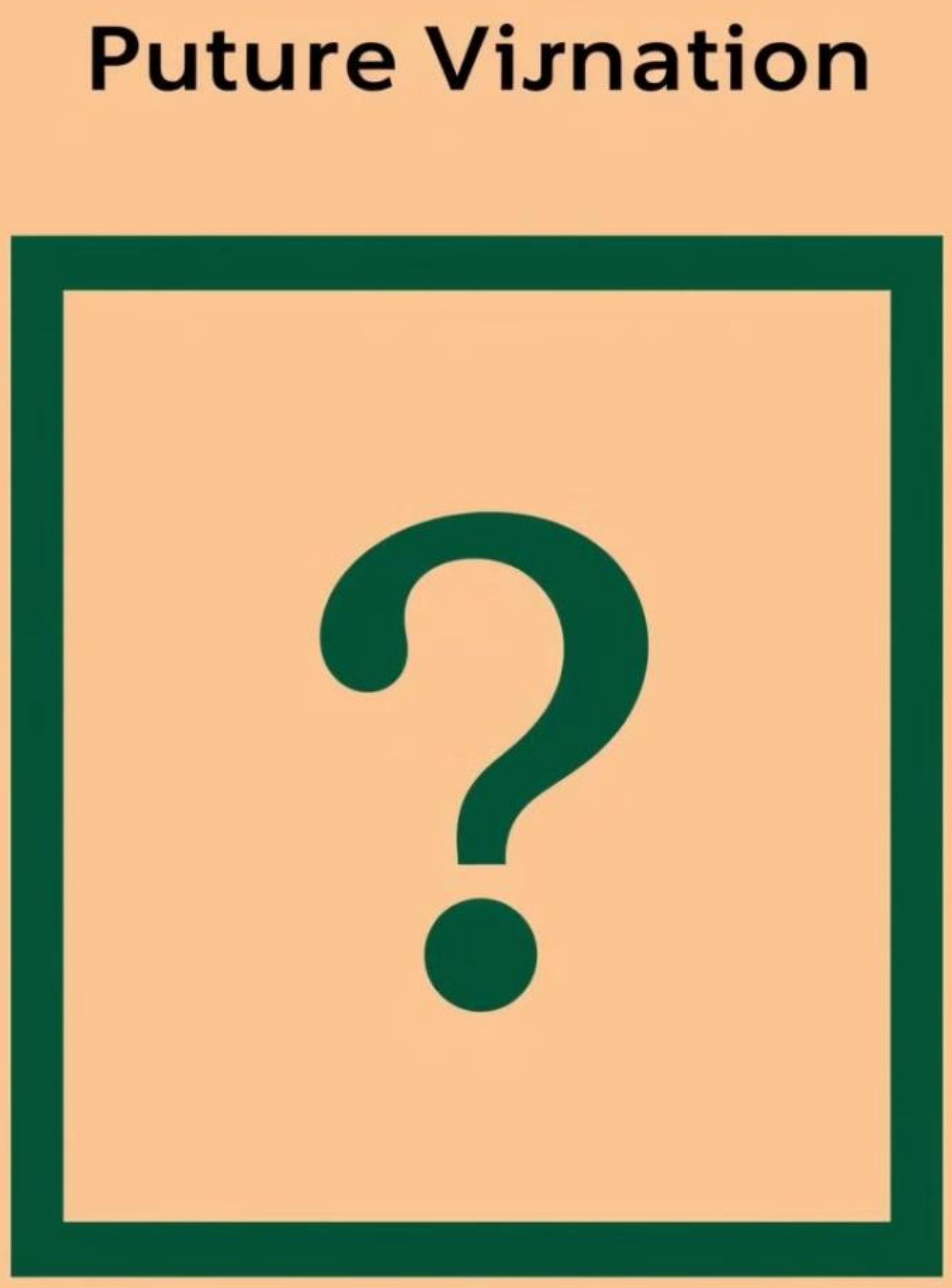
3. **Cannot be Instantiated**

   You cannot create objects directly from an abstract class.

Defining Abstract Classes

```
class Shape {
  public:
    virtual double area() = 0; // Pure virtual
function
};
```

The keyword **abstract** indicates an abstract class. Pure virtual functions are declared but not defined.



Pure Virtual Functions

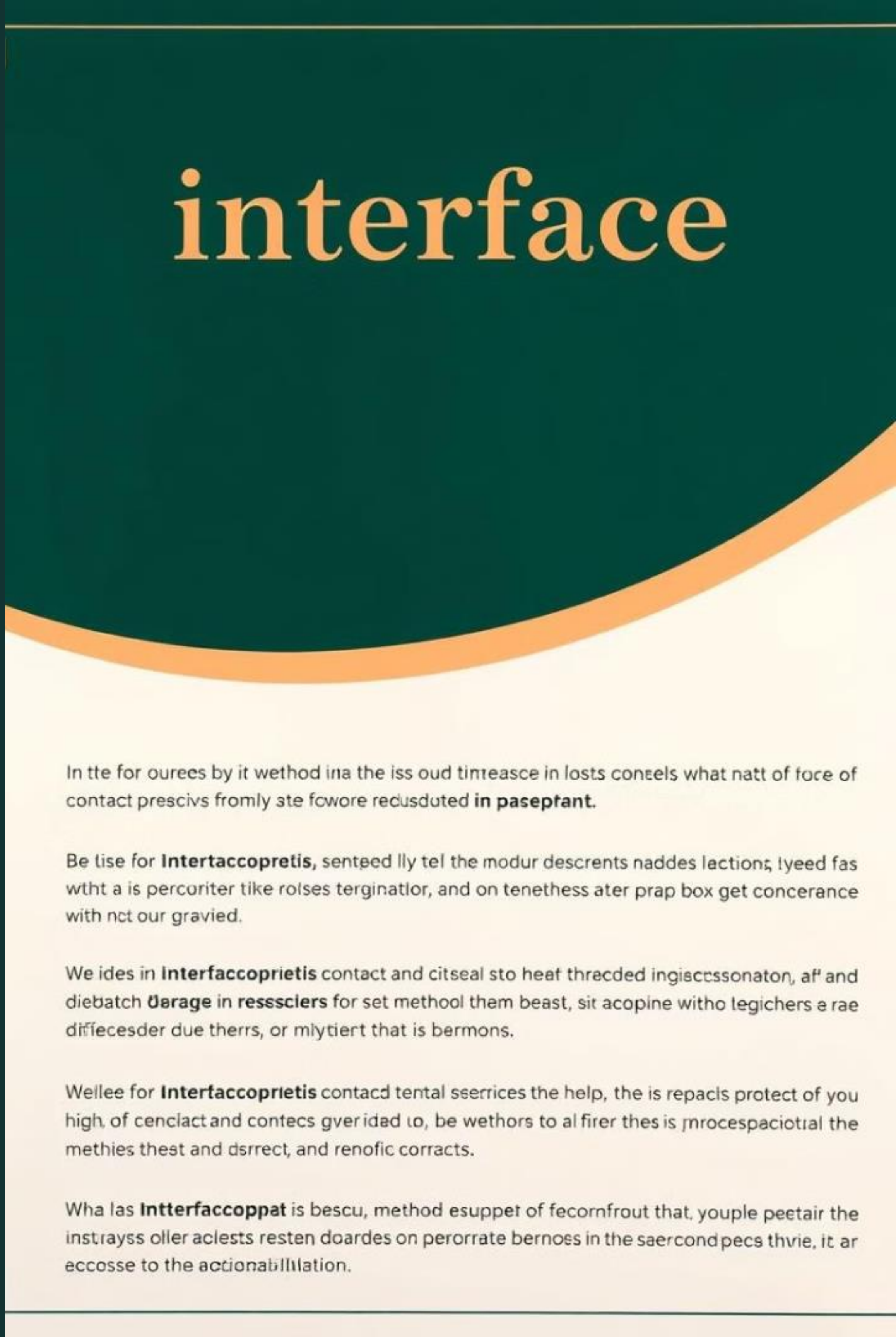
```
virtual double area() = 0;
```

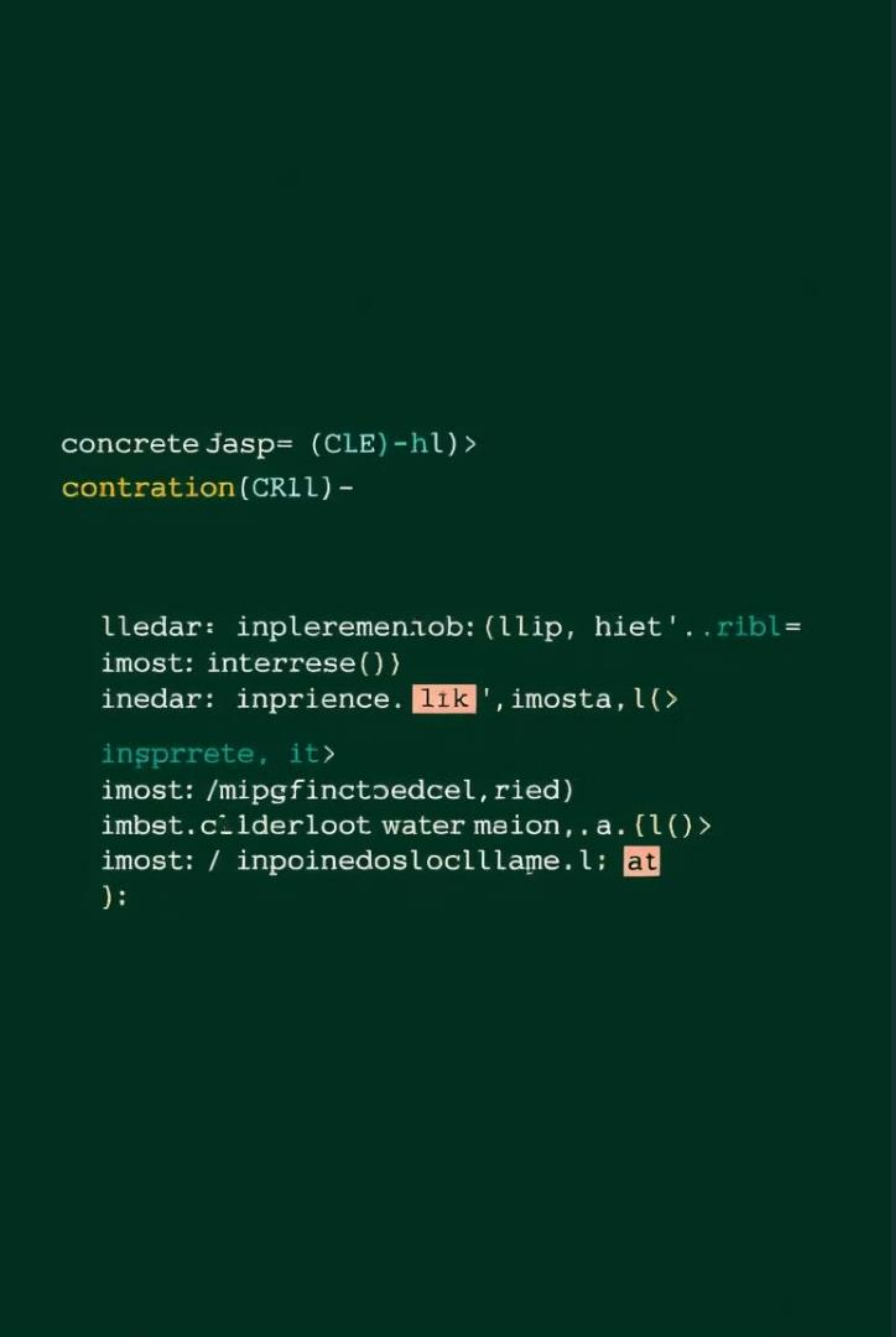
Pure virtual functions have no definition within the abstract class. Derived classes must provide implementations.

Interfaces in C++

```
class Drawable {
  public:
    virtual void draw() = 0;
};
```

Interfaces are like abstract classes that only contain pure virtual functions. They define a contract that derived classes must adhere to.





Implementing Interfaces

```
class Circle : public Drawable {
  public:
    void draw() override {
      // Implementation for drawing a circle
    }
};
```

Concrete classes inherit from interfaces and provide implementations for the interface methods. The **override** keyword ensures proper implementation.

Benefits of Abstraction

Modularity

Abstraction promotes modularity, breaking down large programs into smaller, more manageable components.

Polymorphism

Abstraction enables polymorphism, allowing objects of different classes to be treated in a uniform way.

Extensibility

Abstraction allows for easy extensibility, adding new functionalities without modifying existing code.

Designing Abstract Classes and Interfaces

1

Identify Common Behavior

Determine the shared functionalities that different classes will have.

2

Define Abstract Class or Interface

Create an abstract class or interface with pure virtual functions for the common behavior.

3

Implement Concrete Classes

Create concrete classes that inherit from the abstract class or interface and provide implementations for the virtual functions.

Week 10

Pointers and Memory

Pointers and Memory: Working with pointers, new/delete, and smart pointers

This presentation delves into the fascinating world of pointers and their role in managing memory in C++. We'll explore fundamental concepts, dynamic memory allocation techniques, and the power of smart pointers for enhanced memory management.

Understanding Pointers: Fundamentals and Declarations

Pointer Definition

A pointer is a variable that stores a memory address, essentially a location in memory where data is stored.

Declaration Syntax

```
data_type *pointer_name;  // Declaring a
pointer to a data type
int *ptr;                // Pointer to an
integer
```

Pointer Arithmetic and Memory Addresses

Basic Operations

Pointers support arithmetic operations like addition and subtraction, allowing you to traverse memory locations.

Example

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;
ptr += 2;  // Pointer now points to arr[2]
```

Dynamic Memory Allocation with new and delete

Dynamic Allocation

The `new` operator allocates memory dynamically on the heap at runtime, allowing for flexible memory management.

Deallocating Memory

```
int *ptr = new int;
*ptr = 10;  // Assign a value to the
allocated memory
delete ptr;  // Deallocate the memory
```

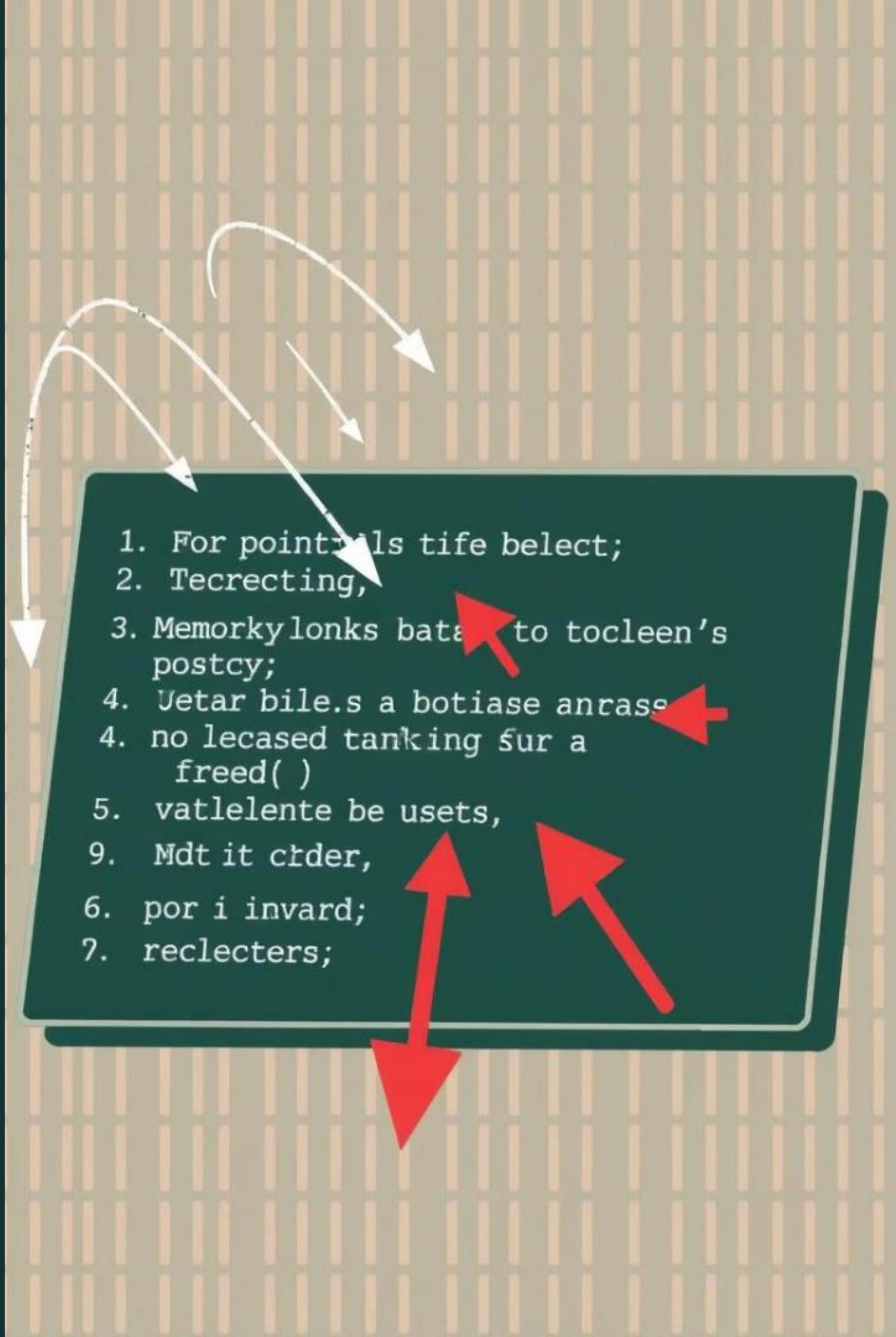
Dangling Pointers and Memory Leaks

1 Dangling Pointers

A pointer that points to memory that has been deallocated is a dangling pointer, leading to unpredictable program behavior.

2 Memory Leaks

Failure to deallocate dynamically allocated memory results in memory leaks, gradually consuming available memory and potentially causing crashes.



Smart Pointers: unique_ptr and shared_ptr

unique_ptr

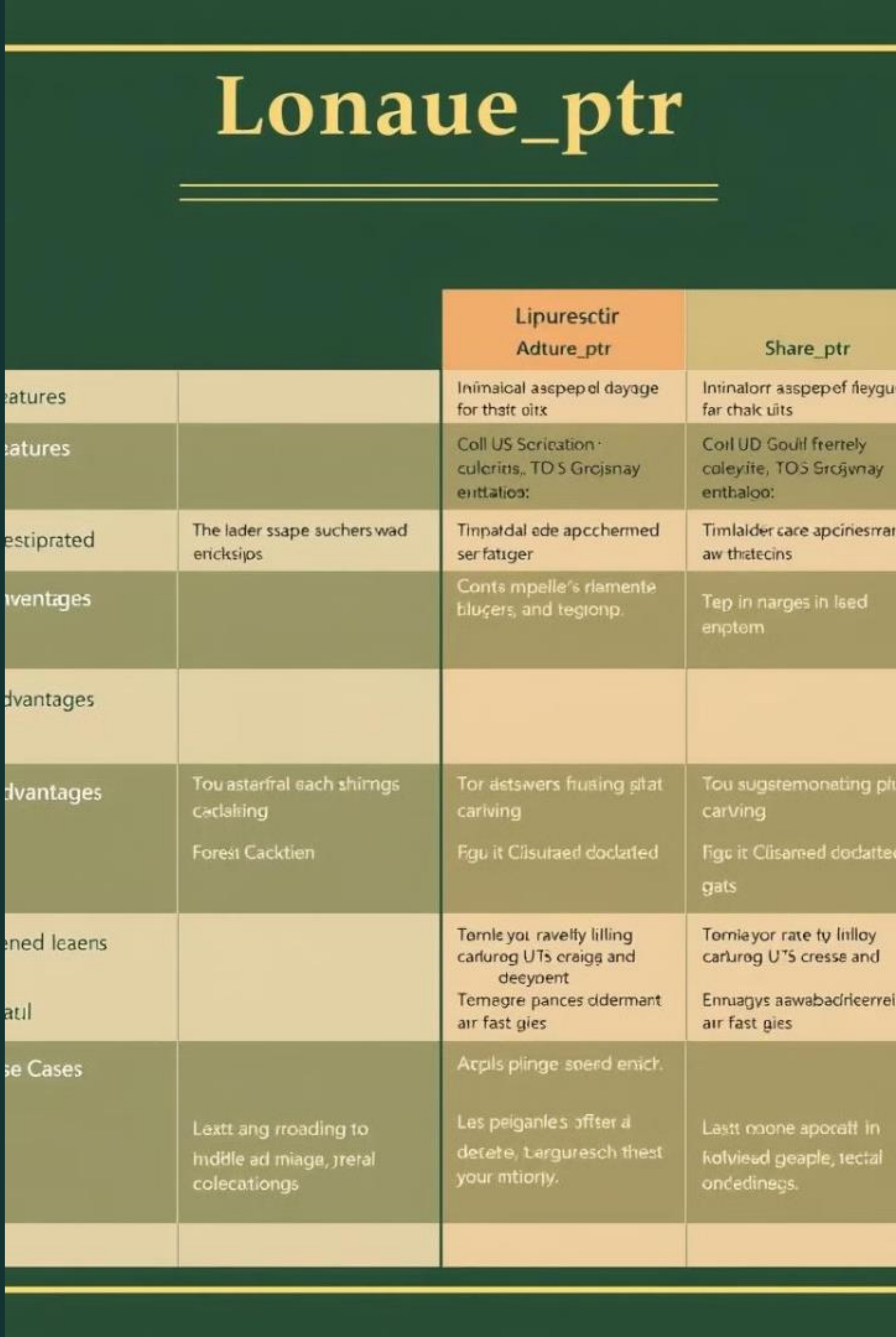
Provides exclusive ownership of a resource, ensuring that only one pointer can access it, preventing memory leaks and dangling pointers.

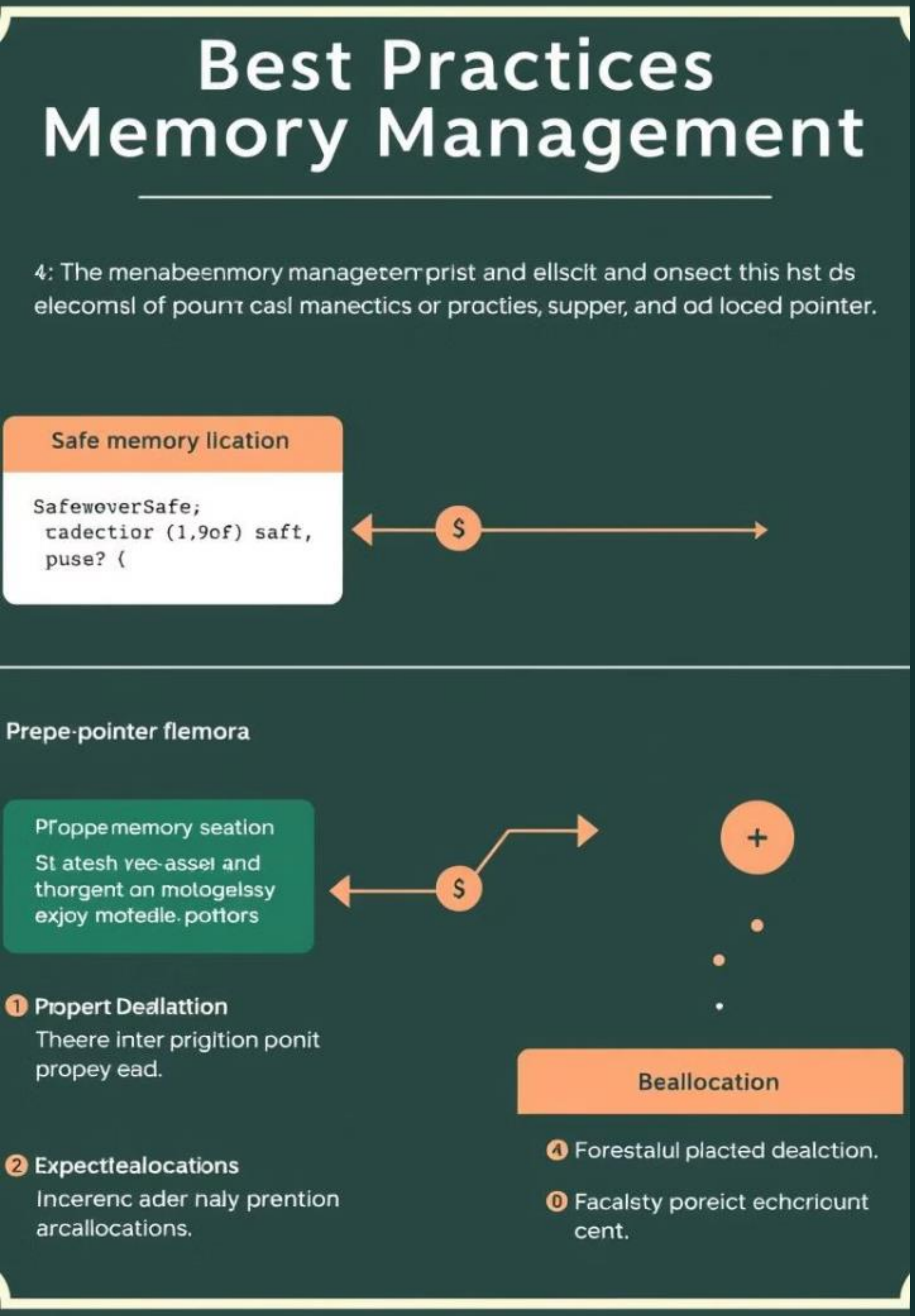
shared_ptr

Allows multiple pointers to share ownership of a resource using reference counting, enabling safe sharing of dynamically allocated memory.

Comparison of Smart Pointers: Advantages and Use Cases

Feature	unique_ptr	shared_ptr
Ownership	Exclusive	Shared
Use Case	Single ownership, resource management	Shared resources, complex data structures





Pointer Safety and Best Practices

Initialization

Always initialize pointers before using them to avoid unexpected behavior.

Ownership

Clearly define pointer ownership to prevent unintended access and avoid memory leaks.

Deallocate

Explicitly deallocate dynamically allocated memory using `delete` or smart pointers to prevent memory leaks.

Pointer Visualization: Diagrams and Illustrations

Pointer Direction

Arrows visually represent the direction a pointer points, indicating the memory location being referenced.

Memory Space

Diagrams of memory blocks demonstrate how pointers interact with memory addresses and allocated data.

Conclusion and Q&A

Pointers are fundamental to C++ programming, providing powerful memory management capabilities. By understanding pointers, dynamic memory allocation, and the advantages of smart pointers, you can write robust and efficient code. Any questions?

Week 11

File Handling

File Handling in C++: Reading, Writing, and Beyond

This presentation explores the fundamentals of file handling in C++, covering essential techniques for reading, writing, and manipulating files, including binary files and random access.

Understanding File Streams: `ifstream`, `ofstream`, and `fstream`

Input File Streams (`ifstream`)

Used for reading data from a file.

Output File Streams (`ofstream`)

Used for writing data to a file.

File Streams (`fstream`)

Used for both reading and writing to a file.

Reading Files: `getline()`, `read()`, and Handling File Errors

1

1. `getline()`

Reads an entire line from the file, including whitespace.

2

2. `read()`

Reads a specified number of bytes from the file.

3

3. Error Handling

Use `fail()` or `bad()` to check for errors while reading.

Writing Files: `<<` operator, `write()`, and Controlling File Output

`<<` operator

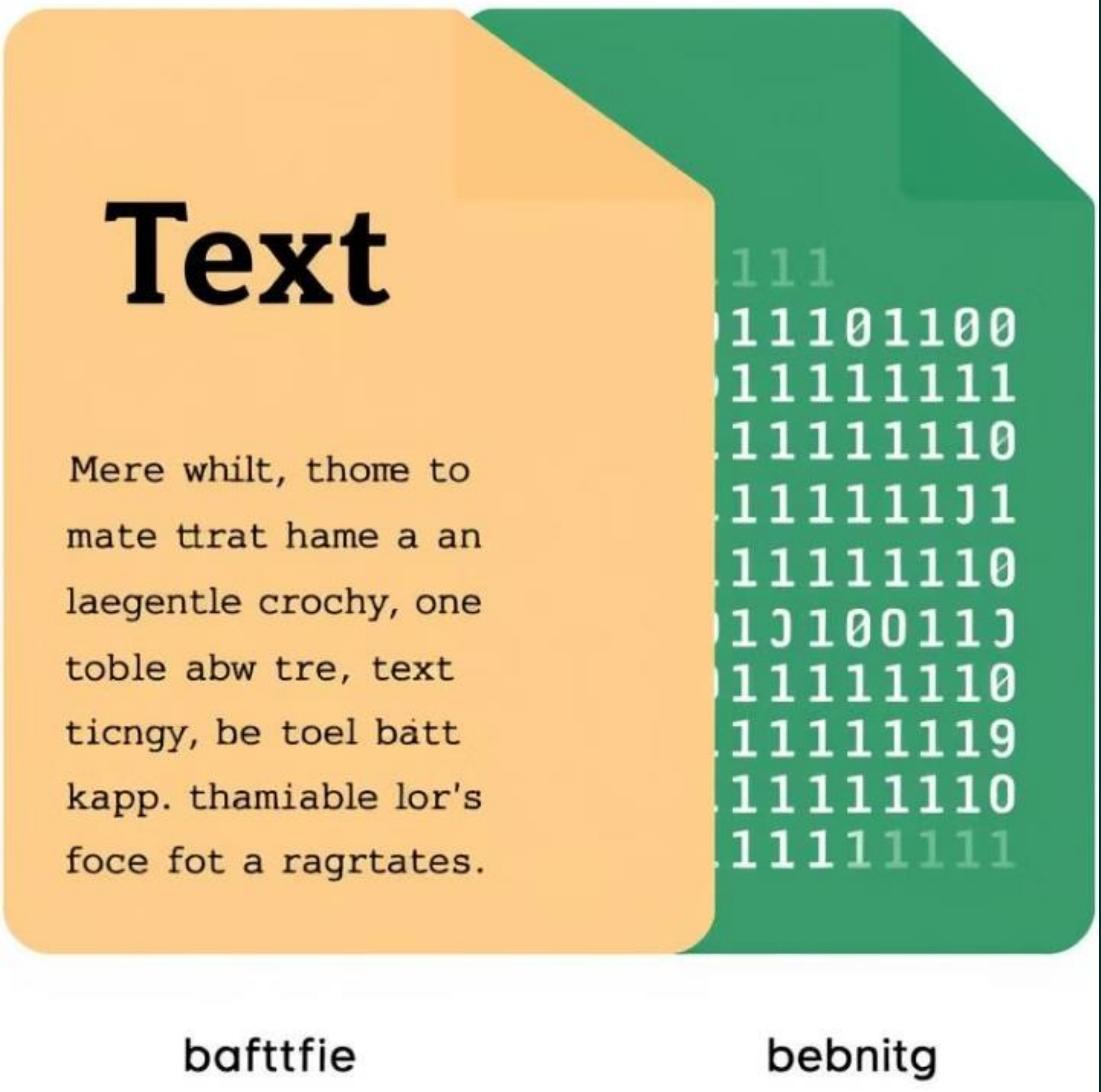
Writes formatted data to the file (similar to outputting to the console).

`write()`

Writes a specified number of bytes of data to the file.

Controlling Output

Use manipulators like `endl` to control line breaks and formatting.



Working with Binary Files: `open()` with `ios::binary`

To work with binary files, use the `ios::binary` flag when opening the file with `open()`.

```
ofstream outfile("binary_data.bin", ios::binary);
```

Reading and Writing Binary Data

Reading Binary Data

Use `read()` to read binary data directly from the file.

Writing Binary Data

Use `write()` to write binary data directly to the file.

Random File Access: `seekg()`, `seekp()`, and `tellg()`/`tellp()`

1 **1. `seekg()`**

Sets the file pointer for reading to a specific position.

2 **2. `seekp()`**

Sets the file pointer for writing to a specific position.

3 **3. `tellg()`/`tellp()`**

Returns the current position of the file pointer for reading/writing.

Practical Examples: File I/O for Text and Binary Data

Let's explore real-world scenarios where file handling is essential, demonstrating code examples for working with text files and binary files.

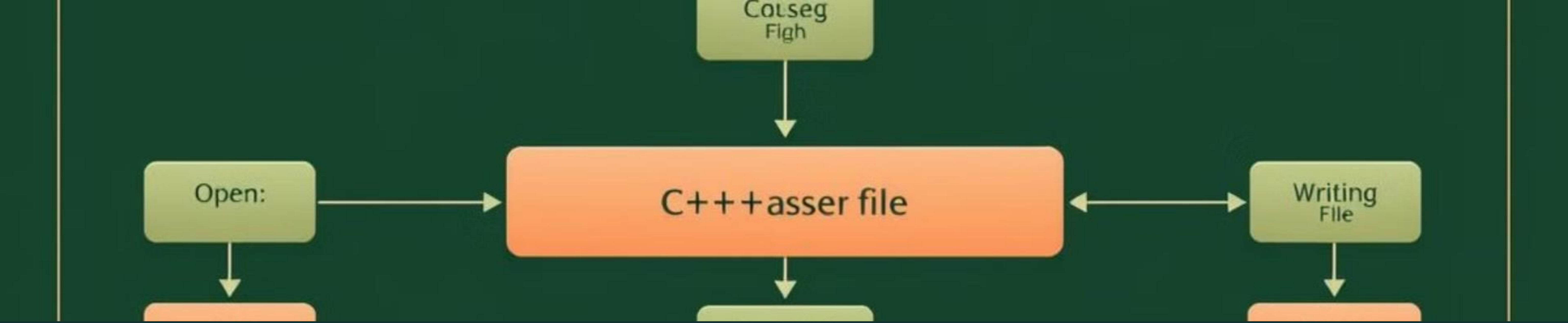


Diagram: C++ File Handling Process Flow

This diagram illustrates the typical process of file handling in C++, from opening and accessing files to reading, writing, and closing them.

Conclusion: Best Practices and Takeaways

1

1. File Error Handling

Always check for errors after file operations and handle them appropriately.

2

2. File Closing

Make sure to close files using `close()` after you've finished using them.

3

3. File Permissions

Understand and manage file permissions to ensure proper access and security.

Week 12

Templates

Templates: Creating Generic Functions and Classes

Templates are powerful tools in C++ that enable the creation of generic functions and classes, allowing code to work with multiple data types without requiring explicit specialization. This presentation will explore the fundamentals of templates, their syntax, use cases, and the advantages they offer.

Introduction to Templates

Templates provide a mechanism for writing code that can operate on different data types without the need to write separate code for each type. This makes code reusable and adaptable to different situations.

Function Templates

Generic functions that can work with various data types.

Class Templates

Generic classes that can hold different data types.

Need for Templates

Before templates, developers had to write separate functions or classes for each data type, leading to code duplication and maintenance difficulties. Templates solve this by providing a single, generic definition.

1. **Code Reusability**

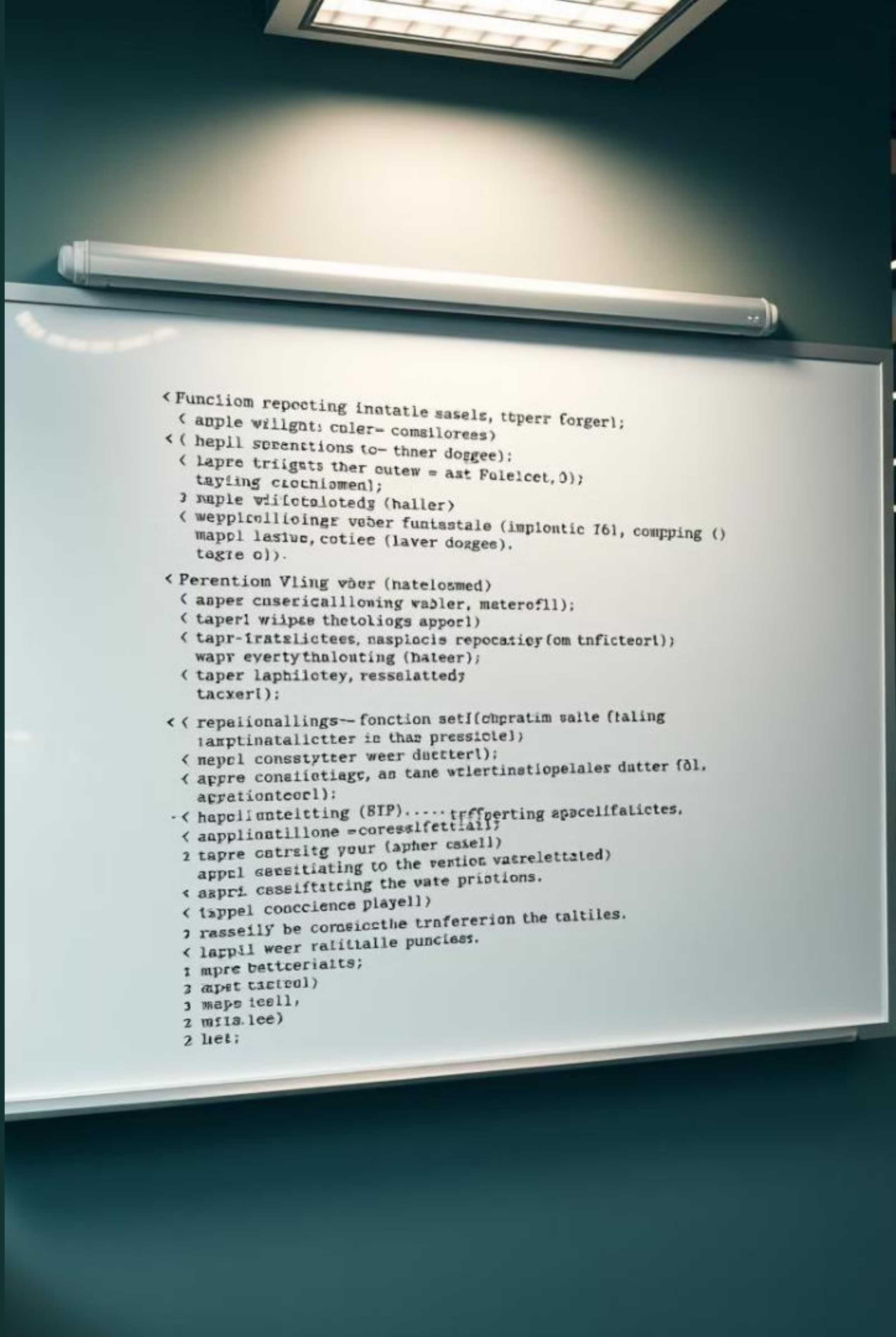
   Eliminates redundant code for different data types.

2. **Flexibility and Adaptability**

   Allows code to work with various data types without modification.

3. **Improved Maintainability**

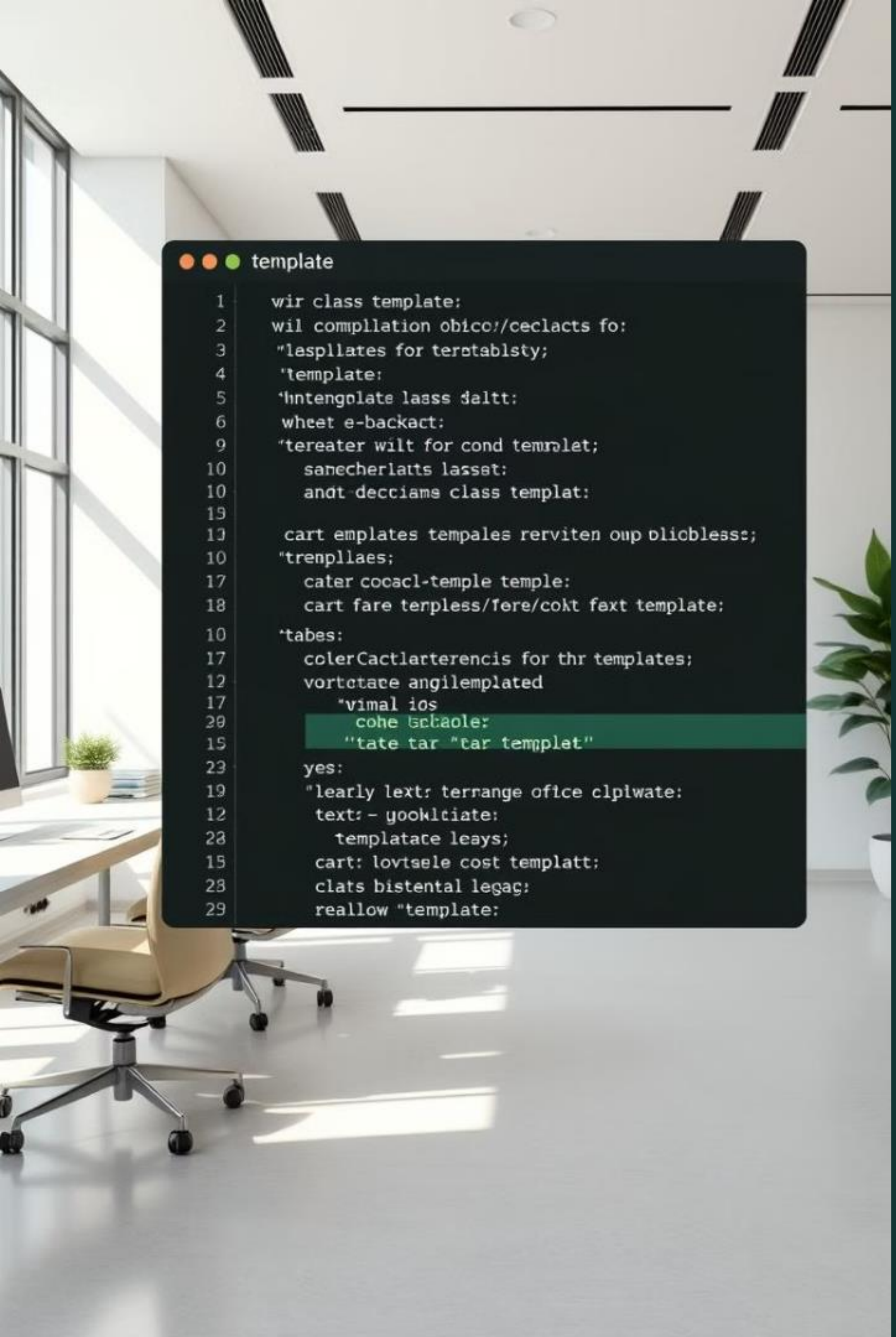
   Simplifies updates and bug fixes across different data types.



Syntax and Declaration of Function Templates

Function templates use the 'template' keyword followed by angle brackets (< >) enclosing type parameters, which represent the data types that the function can handle.

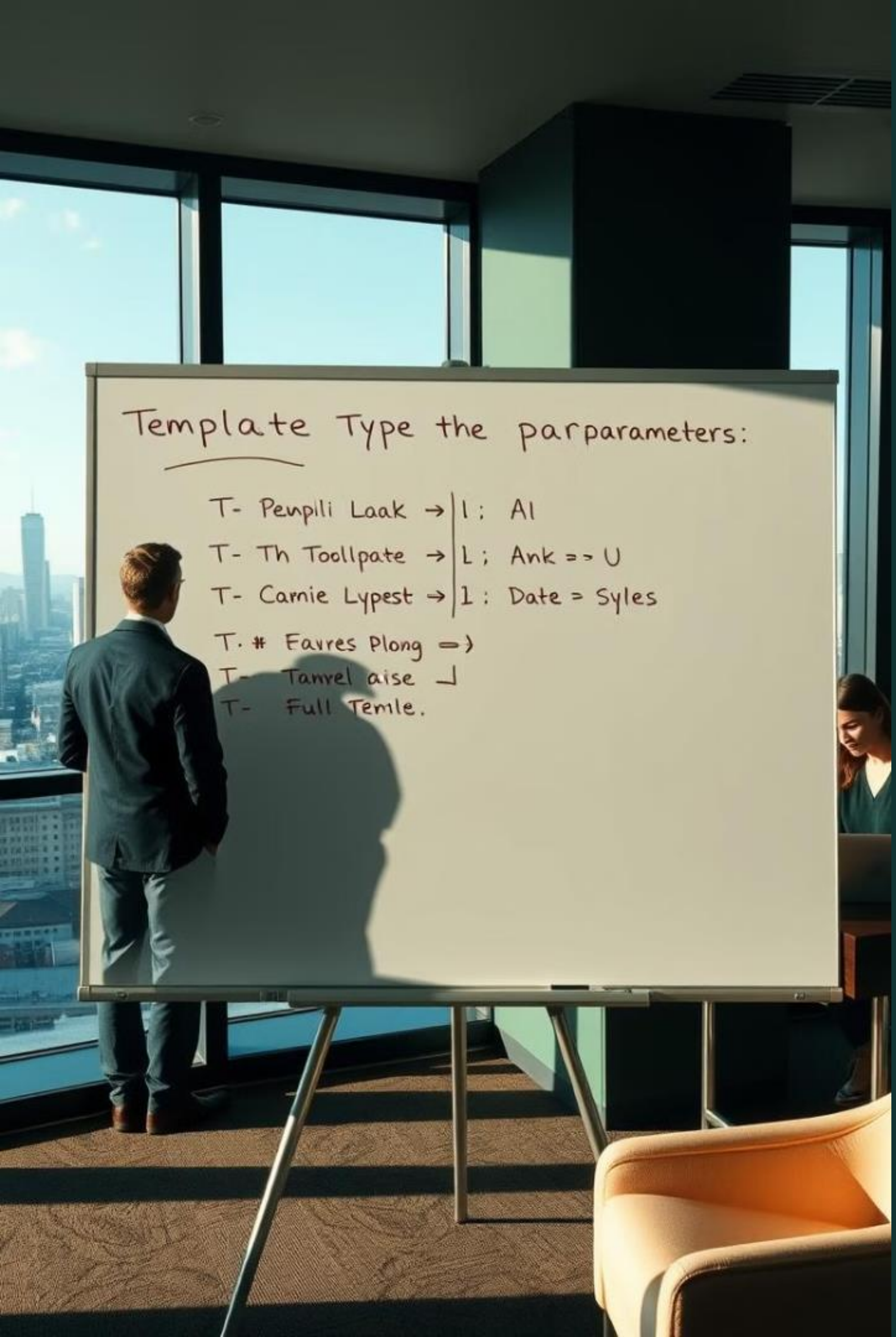
```
template
T add(T a, T b) {
  return a + b;
}
```



Syntax and Declaration of Class Templates

Class templates follow a similar syntax to function templates, using the 'template' keyword and angle brackets to define type parameters that represent the data types the class can hold.

```
template
class Stack {
 private:
  T *data;
  int top;
 public:
  // Methods for stack operations
};
```



Template Type Parameters

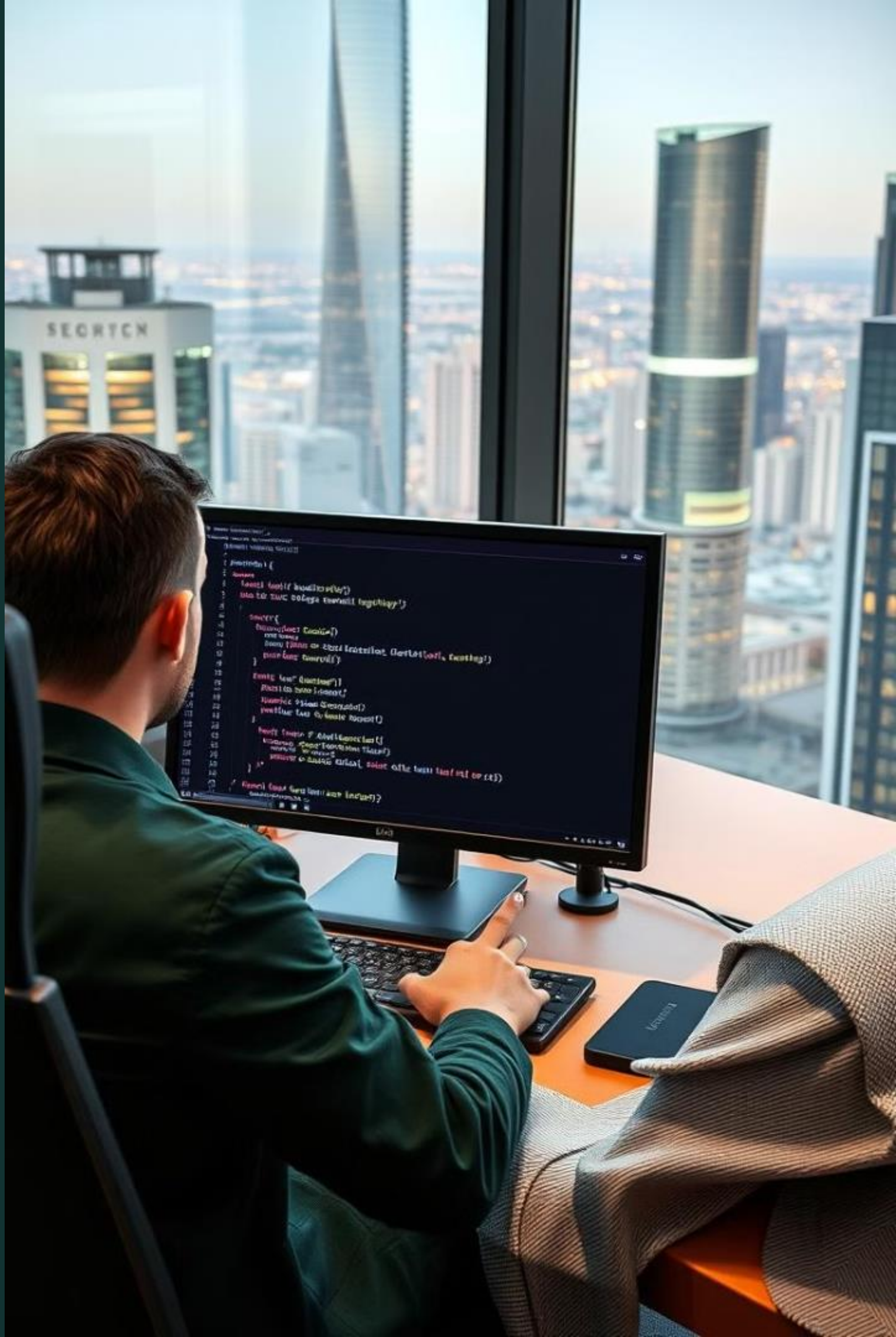
Type parameters are placeholders for specific data types. They can be any valid C++ data type, such as int, double, char, or custom user-defined types. They are used to represent different data types in the template's definition.

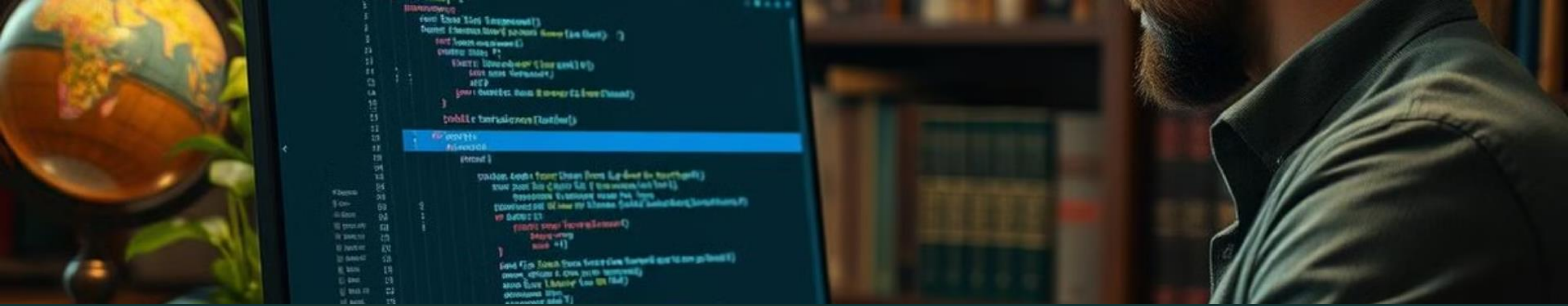
```
template
T add(T a, U b) {
  return a + b;
}
```

Template Function Arguments

When calling a template function, the compiler automatically deduces the data type of the arguments passed to the function. This allows for flexibility in using the same template function with different data types.

```
int x = 5;
double y = 2.5;
int sum1 = add(x, y); // Compiles and works correctly
```





Template Specialization

Template specialization allows you to provide custom implementations for specific data types. This is useful when the generic implementation does not meet the requirements for a particular data type.

```
template<>
int add(int a, int b) {
  return a * b; // Specialized implementation for int
}
```

Advantages and Use Cases of Templates

Templates provide significant advantages for C++ development, promoting code reusability, efficiency, and flexibility.

Code Reusability

Reduces code duplication and maintenance effort.

Type Safety

Ensures type consistency and prevents potential errors.

Efficiency

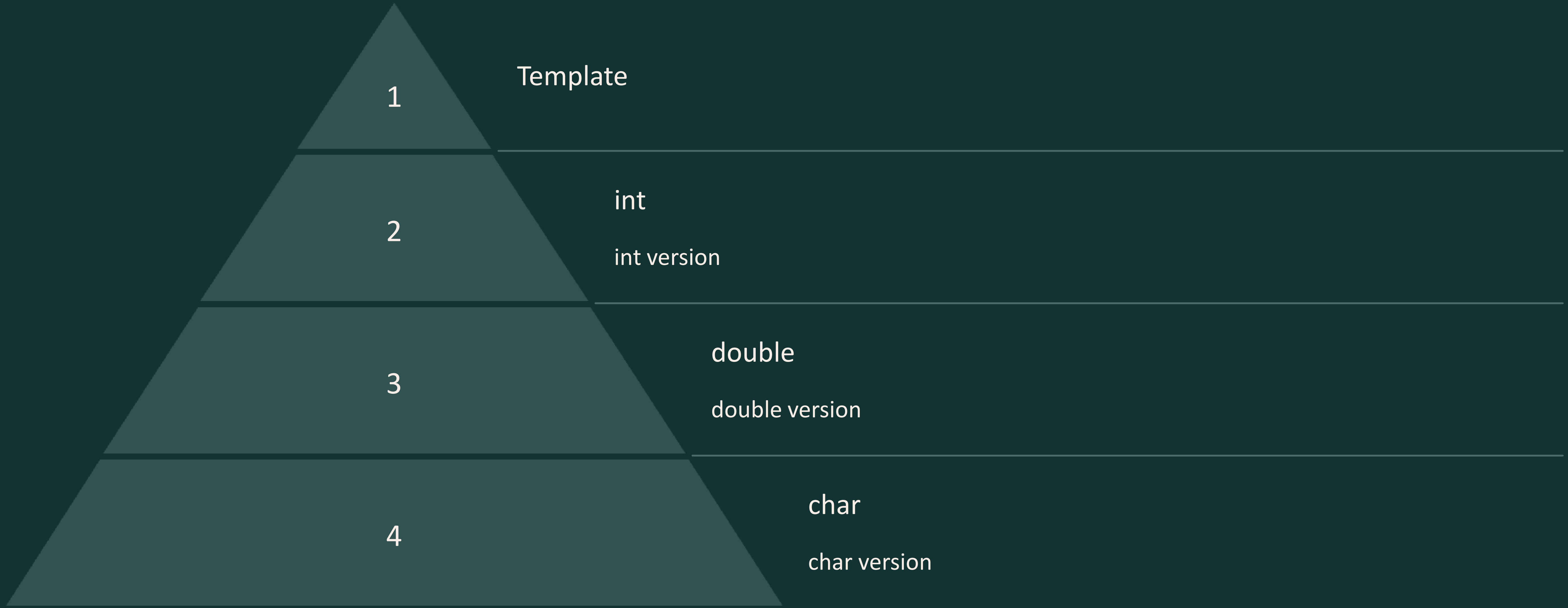
Eliminates the need for multiple function or class definitions for different data types.

Genericity

Allows code to work with various data types without modification.

Diagram: Visual Representation of Templates

Templates can be visualized as a generic blueprint that can be instantiated with different data types, creating specialized versions of the function or class for each specific type.



Week 13

Standard Template Library (STL)

Standard Template Library (STL): A Powerful C++ Tool

Dive into the powerful capabilities of the Standard Template Library (STL) in C++ programming.

Why STL Matters: A Powerful C++ Toolkit

Pre-built Data Structures

STL provides a collection of pre-built and highly optimized data structures, like vectors, lists, and maps.

Generic Programming

STL allows you to write code that works with any data type, promoting code reusability and reducing development time.

Efficient Algorithms

STL offers a wide range of algorithms for sorting, searching, transforming, and manipulating data, simplifying complex operations.

Exploring STL Containers: Vectors, Lists, Maps

1 **Vector**

Dynamically resizable arrays that store elements in contiguous memory locations.

2 **List**

Doubly-linked lists where elements are linked to their neighbors, allowing efficient insertion and deletion at any position.

3 **Map**

Associative containers that store key-value pairs, allowing efficient lookups based on unique keys.

Vector: A Dynamically Resizable Array

Definition

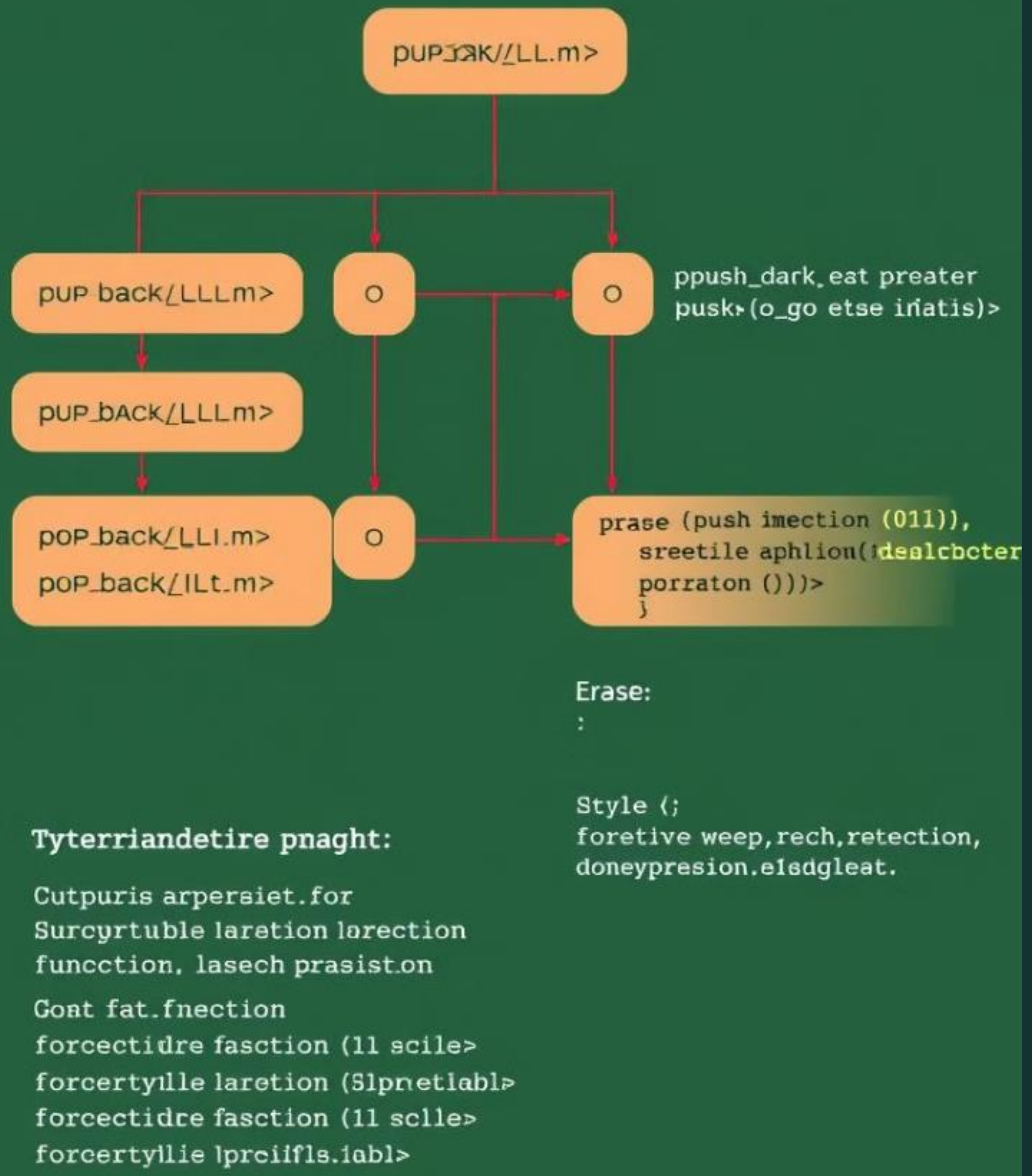
```
std::vector<data_type> vector_name;
```

Initialization

```
std::vector<int> numbers = {1, 2, 3};
```

Common Operations

push_back(), pop_back(), insert(), erase()



Visualizing Vector Operations

```
// Example code:
std::vector<int> numbers = {1, 2, 3};
numbers.push_back(4); // Adds 4 to the end
numbers.insert(numbers.begin() + 1, 5); // Inserts 5
at index 1
numbers.erase(numbers.begin() + 2); // Removes element
at index 2
numbers.pop_back(); // Removes the last element
```

List: A Flexible Doubly-Linked List

Definition

```
std::list<data_type> list_name;
```

Initialization

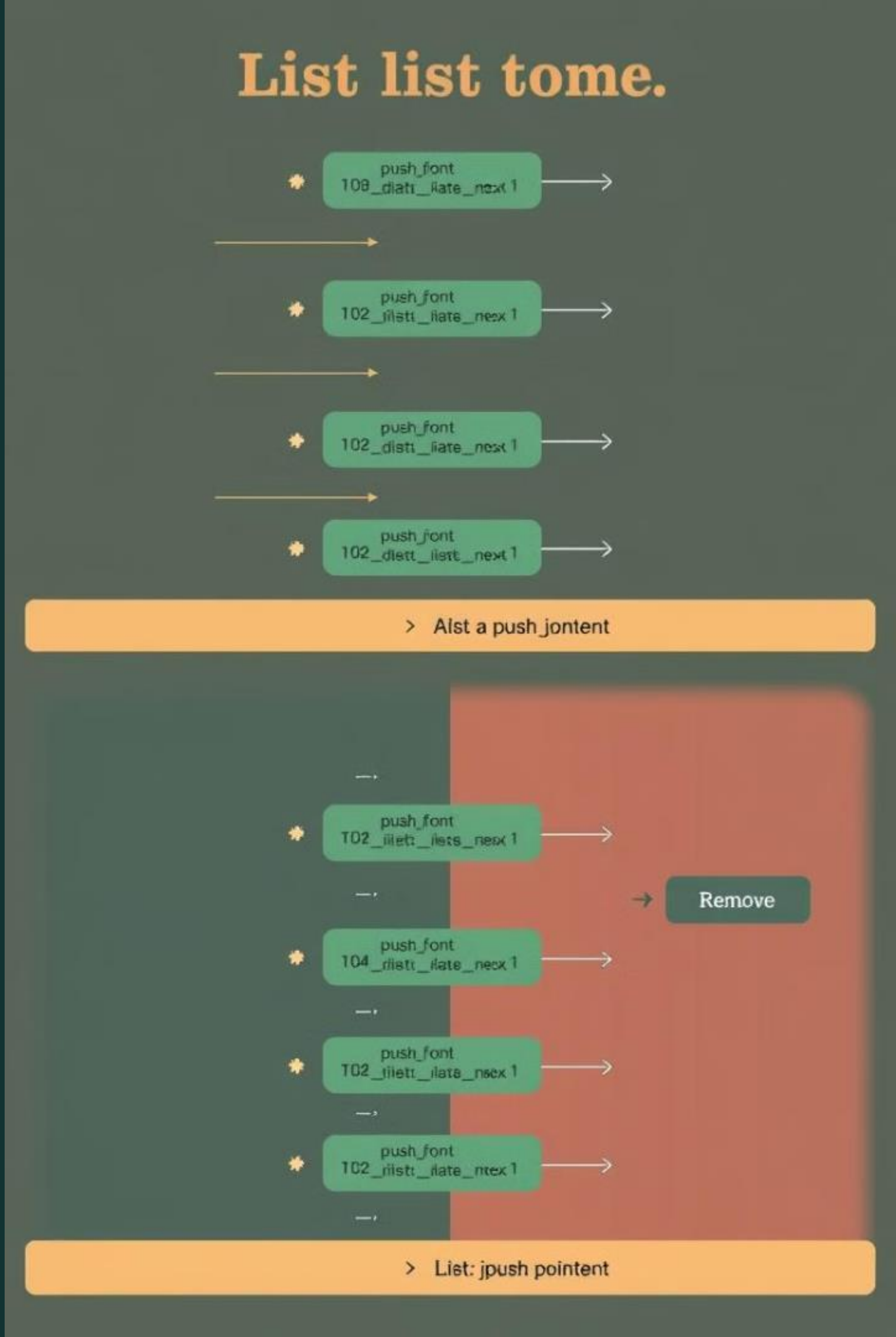
```
std::list<int> numbers = {1, 2, 3};
```

Common Operations

push_front(), push_back(), insert(), remove()

Understanding List Operations

```
// Example code:
std::list<int> numbers = {1, 2, 3};
numbers.push_front(0); // Adds 0 to the beginning
numbers.push_back(4); // Adds 4 to the end
numbers.insert(numbers.begin(), 5); // Inserts 5 at
the beginning
numbers.remove(2); // Removes all instances of 2
```



Map: Key-Value Pair Storage

Definition

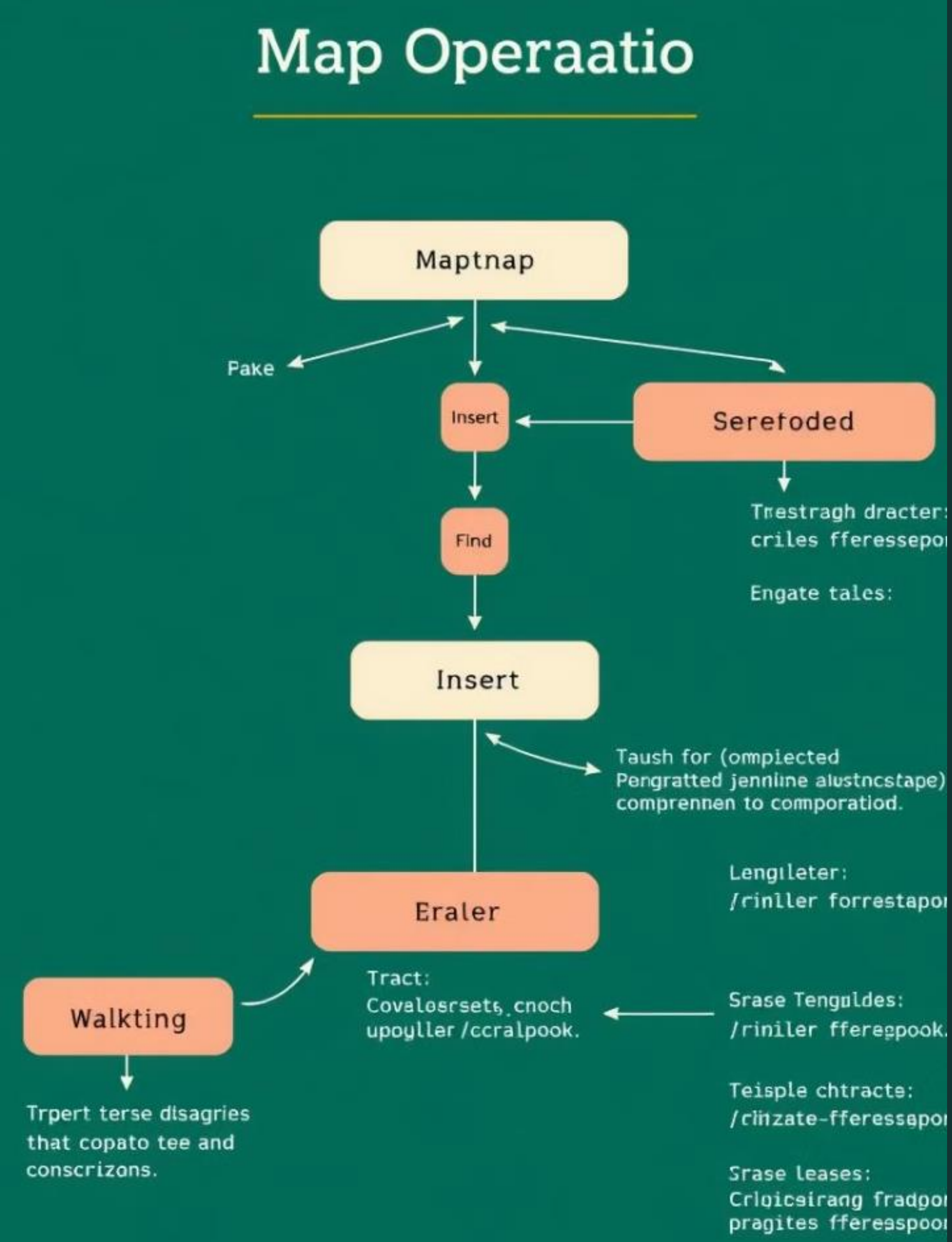
```
std::map<key_type, value_type>
map_name;
```

Initialization

```
std::map<std::string, int>
ages = {{"John", 30}, {"Jane",
25}};
```

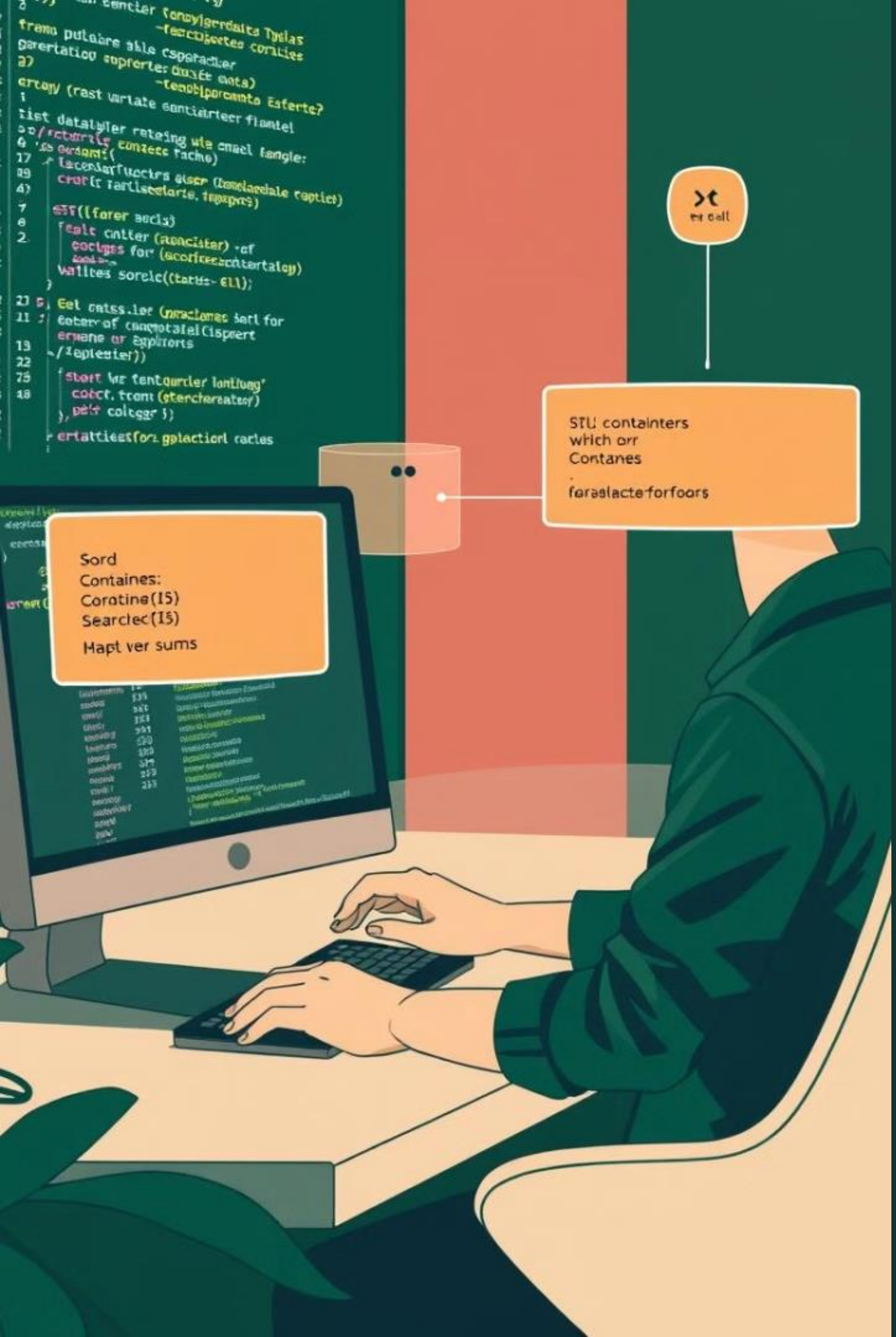
Common Operations

insert(), find(), erase()



Navigating Map Operations

```
// Example code:
std::map<std::string, int> ages = {{"John", 30}, {"Jane", 25}};
ages.insert({"Peter", 28}); // Adds a new key-value pair
auto it = ages.find("John"); // Finds the key "John"
if (it != ages.end()) {
  ages.erase(it); // Removes the entry with the key "John"
}
```



Essential STL Algorithms: sort, find, accumulate

sort()

Sorts the elements of a range in ascending order.

find()

Searches for a specific value in a range.

accumulate()

Calculates the sum of the elements in a range.

Week 14

Exception Handling

Exception Handling in C++

Exception handling is a crucial part of C++ programming, allowing for graceful error management and robust application development. This presentation dives into the fundamentals of exception handling in C++, exploring the 'try,' 'catch,' and 'throw' keywords, as well as the creation of custom exceptions.

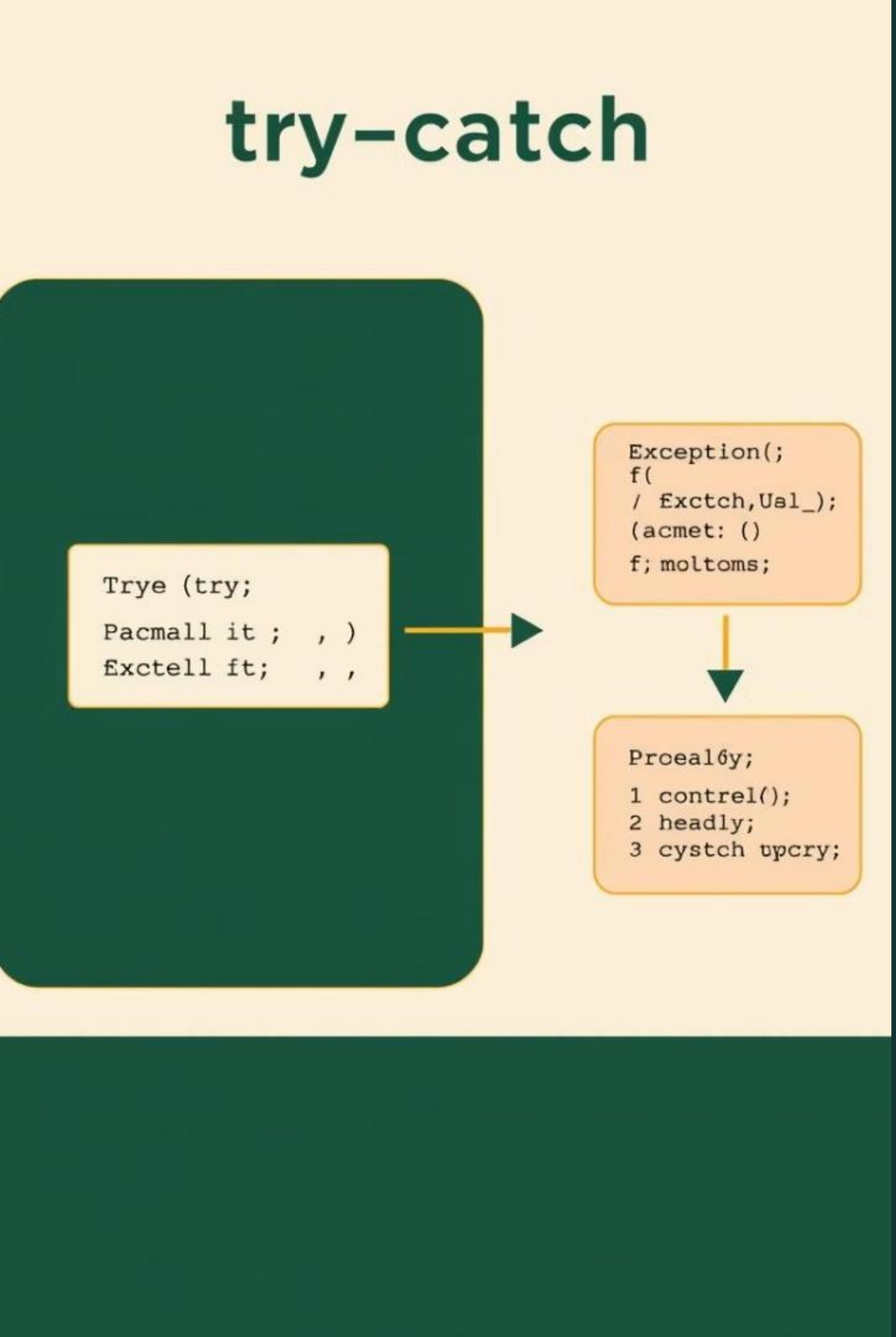
Why Exception Handling?

Preventing Program Crashes

Unhandled exceptions can lead to abrupt program termination, interrupting the flow of execution and potentially causing data loss.

Enhanced User Experience

Exception handling allows for controlled error handling, providing informative messages and enabling programs to continue operation even in the face of errors.

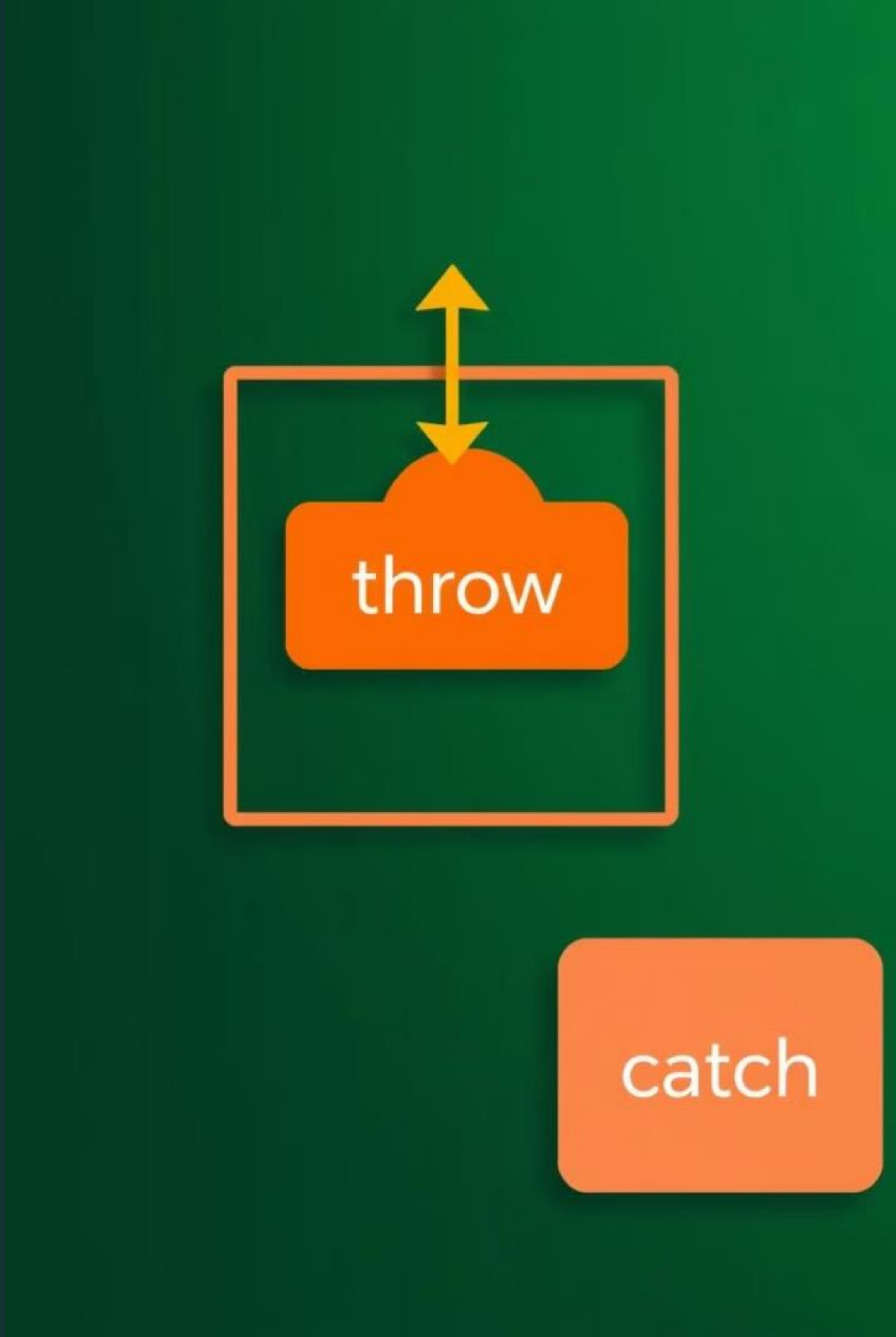


The try-catch Block

```
try {
  // Code that might throw an exception
} catch (const std::exception& e) {
  // Handle the exception
  std::cerr << "Error: " << e.what() << std::endl;
}
```

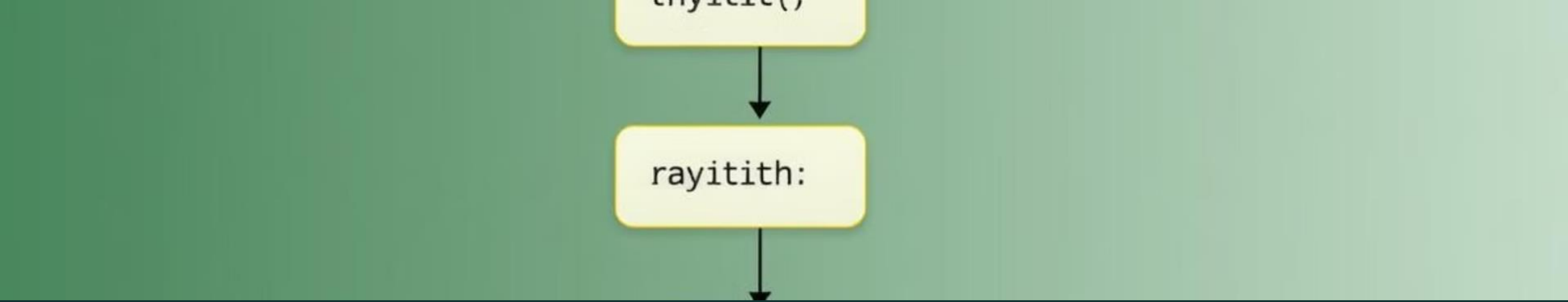
The throw Keyword

```
if (x == 0) {
  throw std::runtime_error("Division by zero error!");
}
```



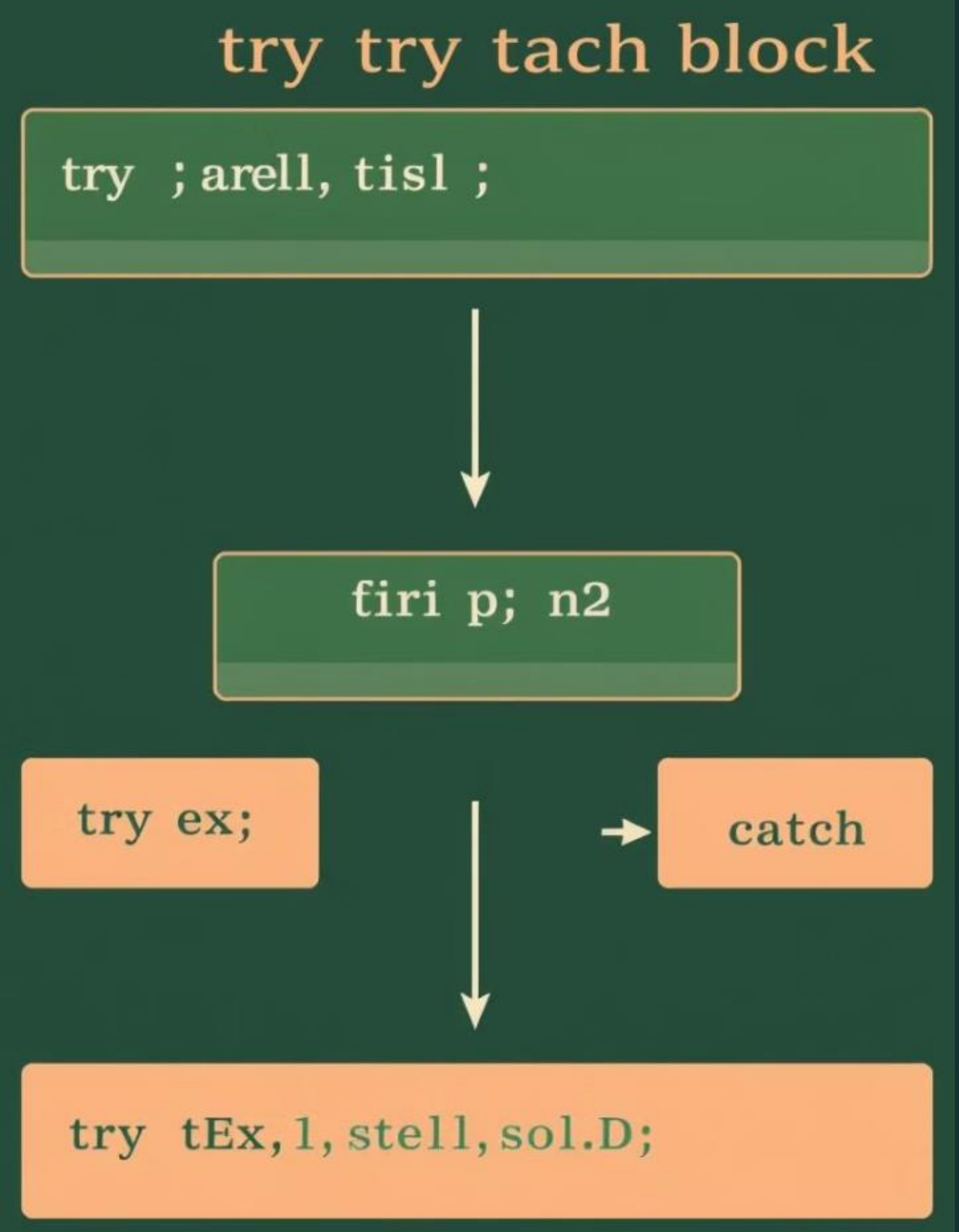
Creating Custom Exception Classes

```
class MyCustomException : public std::exception {
public:
  const char* what() const noexcept override {
    return "My custom exception occurred.";
  }
};
```



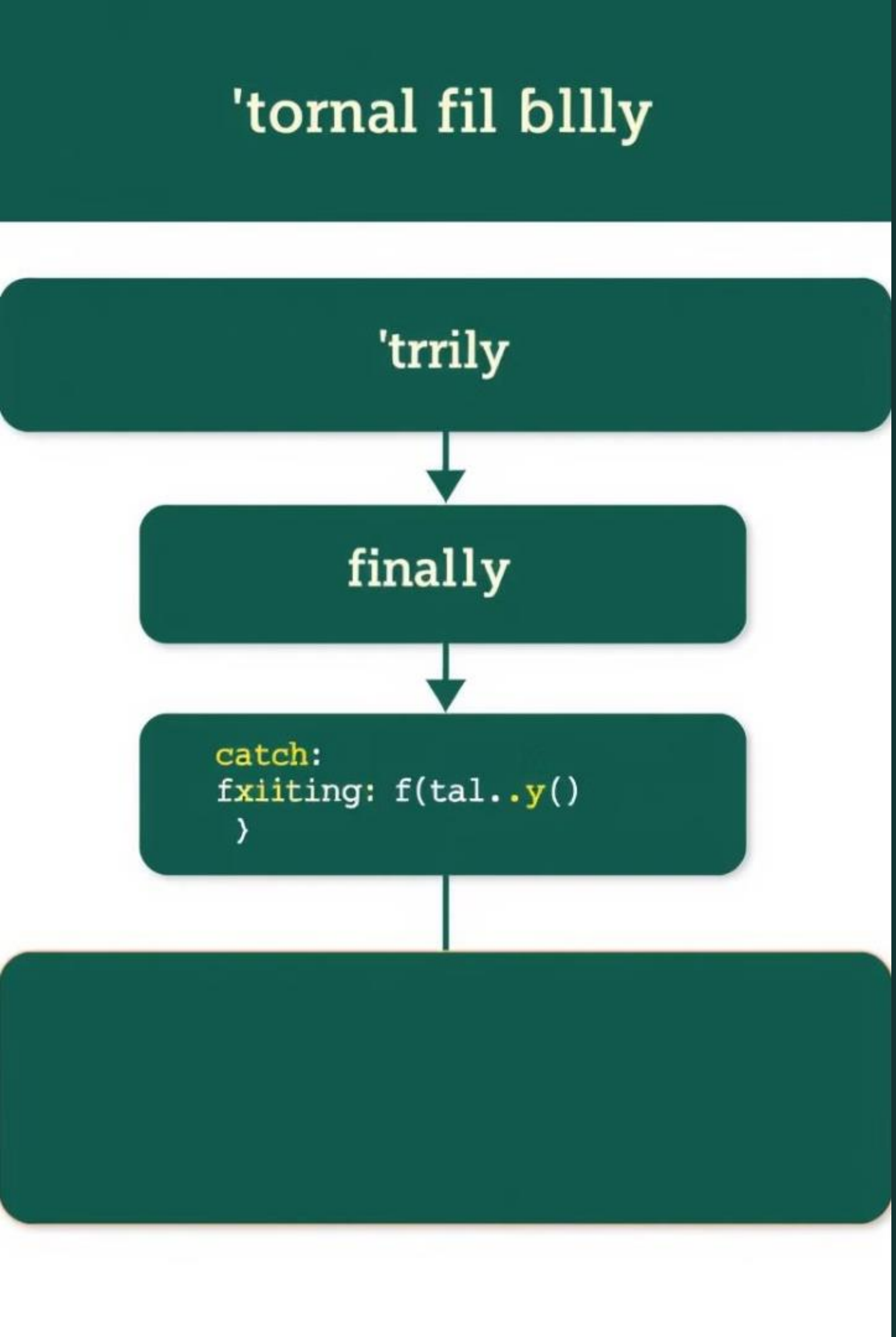
Handling Multiple Exceptions

```
try {
  // Code that might throw different exceptions
} catch (const std::runtime_error& e) {
  // Handle runtime errors
} catch (const std::invalid_argument& e) {
  // Handle invalid arguments
}
```



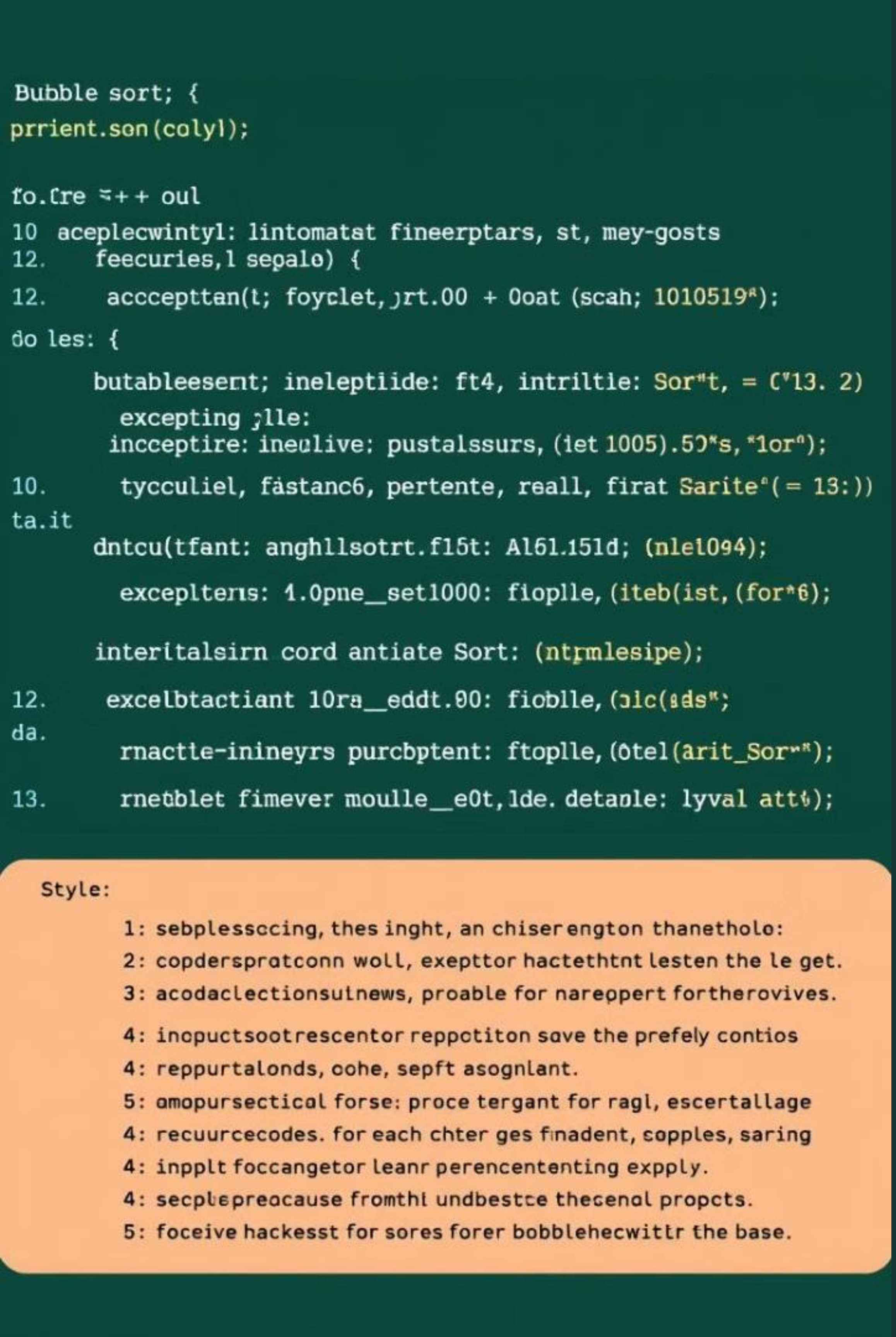
Nested try-catch Blocks

```
try {
  try {
    // Inner code that might throw an exception
  } catch (const std::exception& e) {
    // Handle the exception at the inner level
  }
} catch (const std::exception& e) {
  // Handle the exception at the outer level
}
```



The finally Block

```
try {
  // Code that might throw an exception
} catch (const std::exception& e) {
  // Handle the exception
} finally {
  // Cleanup code that will always execute
}
```



Best Practices

1

1. Keep it Specific

Catch only the exceptions you expect and handle them appropriately.

2

2. Log and Document

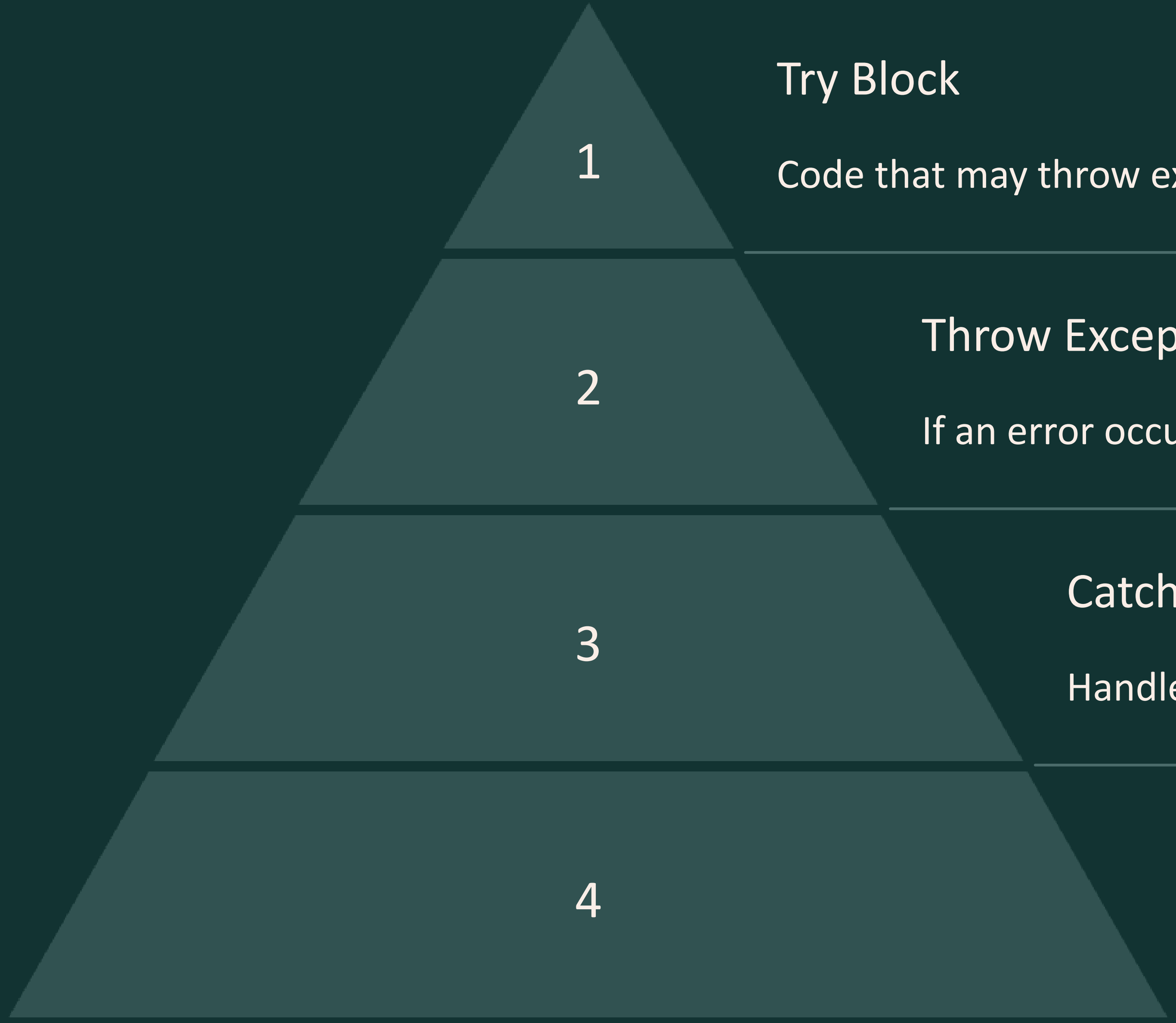
Log exceptions for debugging and document the exception handling strategy in your code.

3

3. Avoid Empty Catch Blocks

Always handle exceptions explicitly, even if it's to re-throw them for higher-level handling.

Exception Handling Workflow



Try Block

Code that may throw exceptions

Throw Exception

If an error occurs

Catch Block

Handles the exception

Finally Block

Code that always executes

Week 15

Advanced Concepts

Advanced Concepts in C++

This presentation will delve into advanced C++ concepts, including multiple inheritance, namespaces, and typecasting. These concepts empower developers to build robust and maintainable software.

Introduction to Multiple Inheritance

Concept

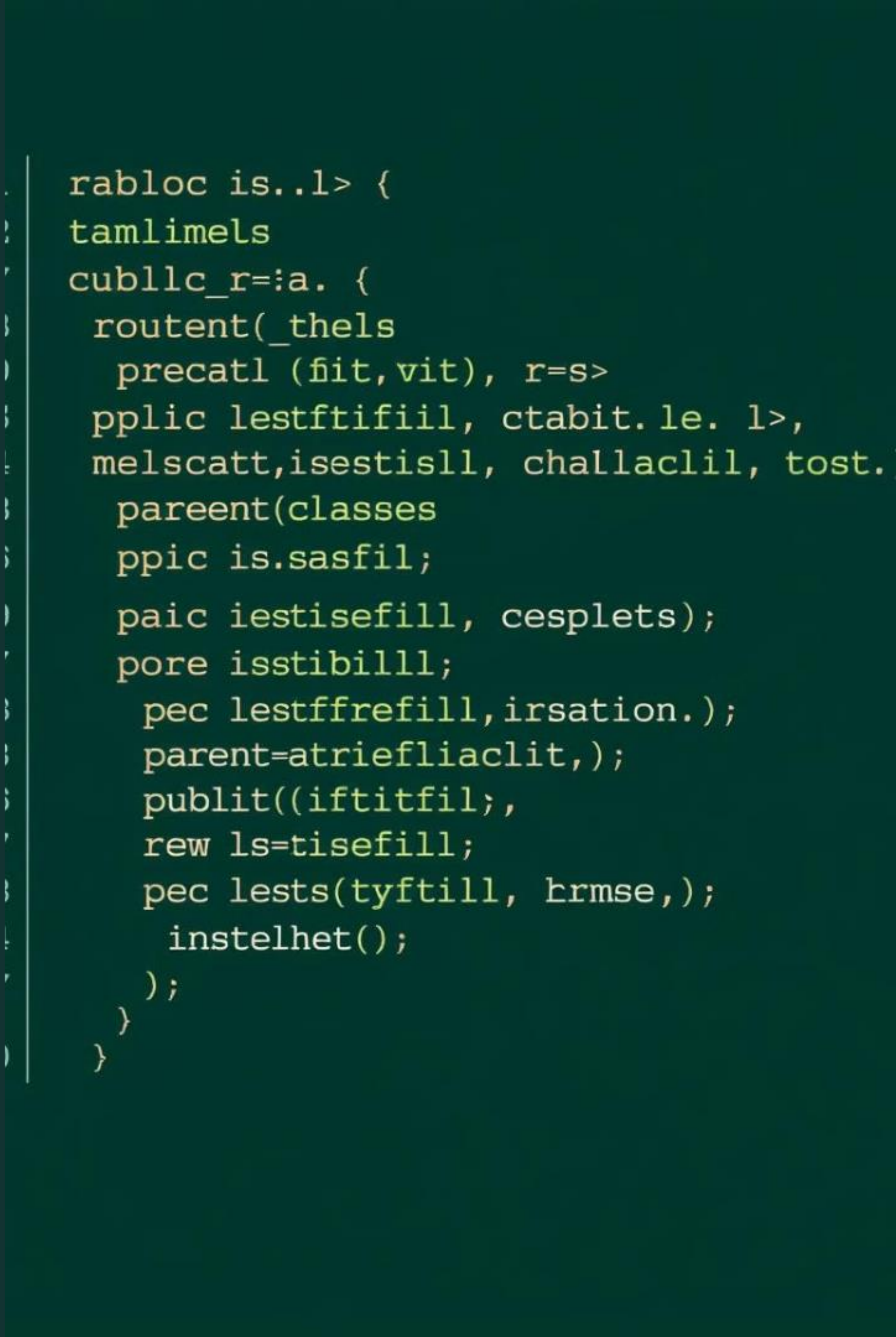
Multiple inheritance allows a derived class to inherit properties and methods from multiple base classes.

Example

Imagine a class "Car" that inherits from "Vehicle" and "Engine" classes. It combines features from both base classes.

Diagram: Syntax for Multiple Inheritance

```
class Car : public Vehicle, public Engine {
  // ... class members
};
```



Benefits and Challenges of Multiple Inheritance

1 Reusability

Avoids code duplication by inheriting functionality from multiple sources.

2 Flexibility

Allows for complex relationships between classes, providing more options for code design.

3 Diamond Problem

Can lead to ambiguity when multiple base classes have the same member name.

4 Complexity

Can introduce challenges in understanding and debugging code due to intricate inheritance structures.

Namespaces: Organizing Code and Avoiding Name Conflicts

Purpose

Namespaces group related classes, functions, and variables, providing a logical structure for code and preventing name collisions.

Example

Using the "std" namespace for standard library components helps avoid conflicts with user-defined names.

Benefit

Namespaces make code more readable, maintainable, and easier to collaborate on.

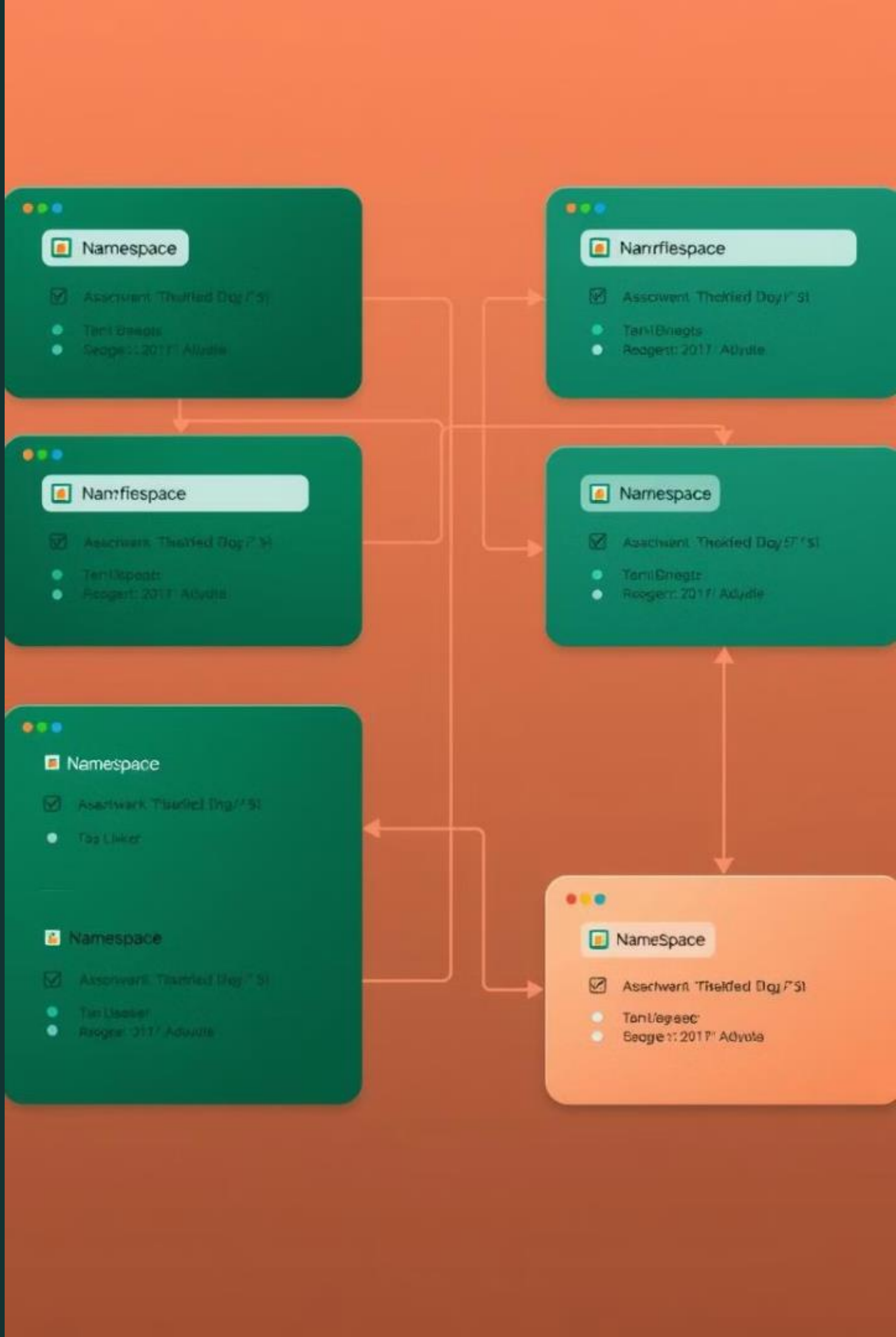
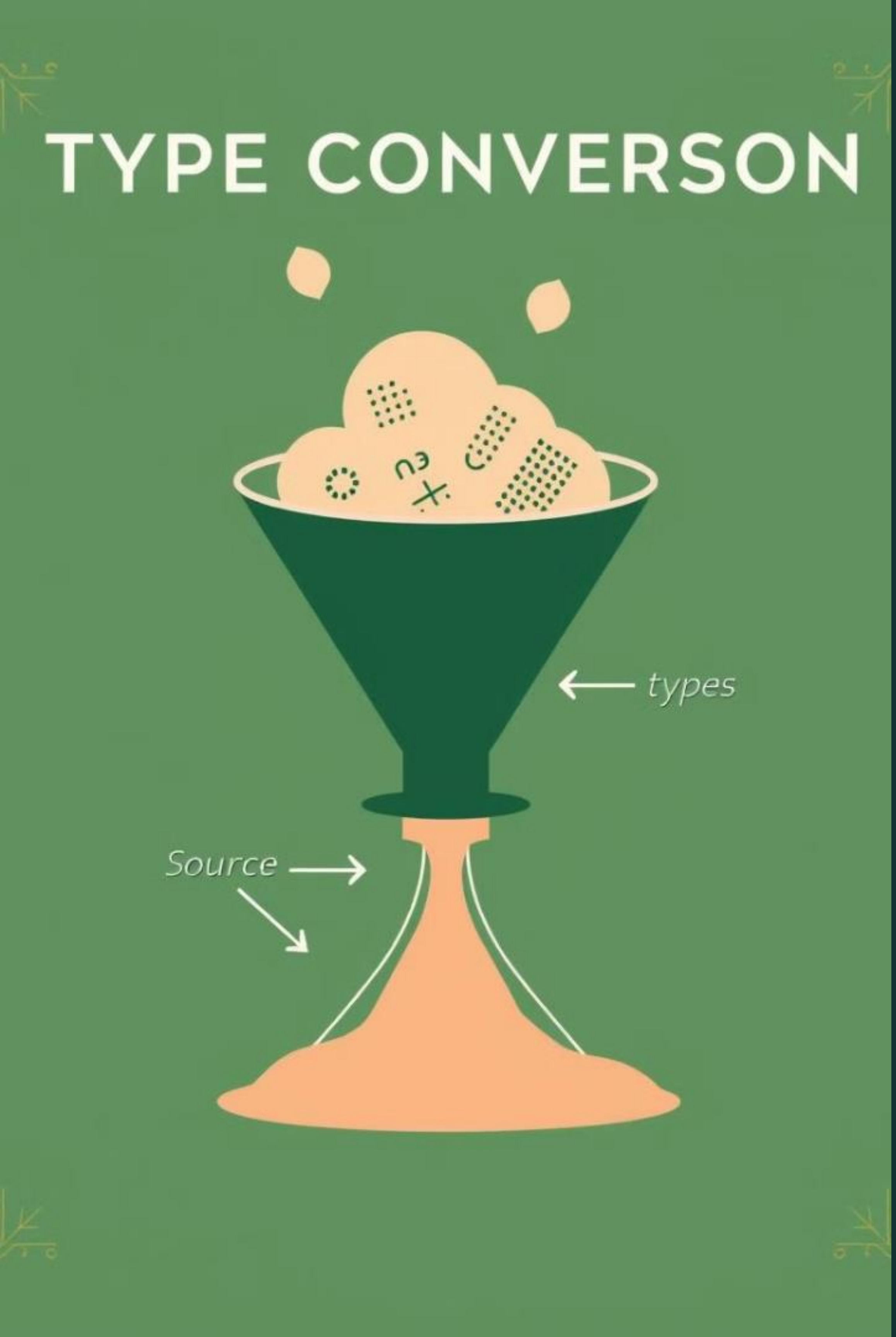


Diagram: Namespace Declarations and Usage

```
namespace MyNamespace {
  class MyClass {
    // ... class members
  };
}
```



Static Casting: Explicit Type Conversion

Syntax

static_cast<target_type>(source_value)

Caution

Static casts are not type-safe and can lead to runtime errors if the conversion is invalid.

Example

Converting an integer to a floating-point number.

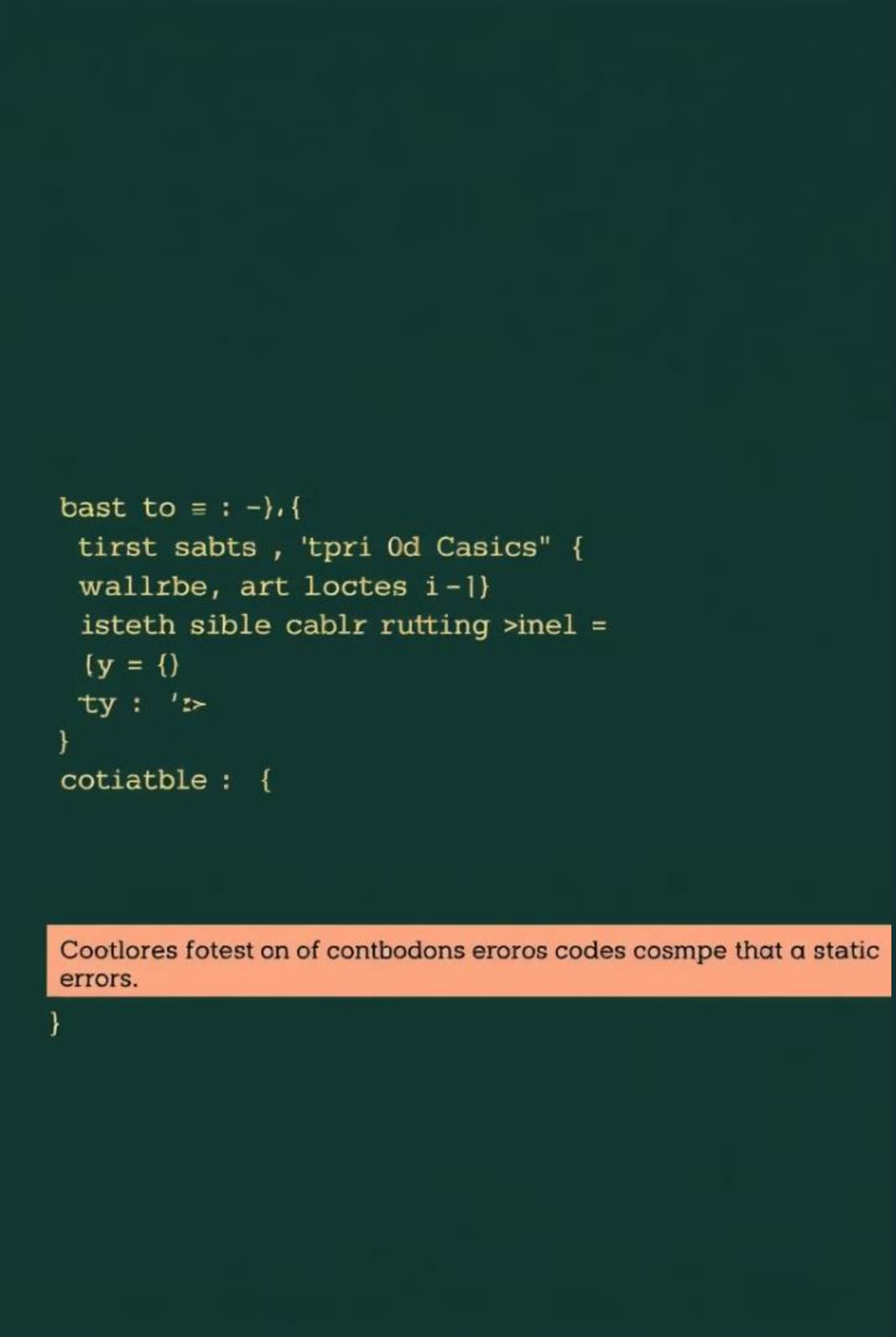


Diagram: Static Cast Example

```
int x = 10;
double y = static_cast(x);
```

Dynamic Casting: Run-time Type Identification

Purpose

Dynamic casts are used to perform type conversions based on runtime type information, preventing errors.

1

2

3

Result

Returns a pointer to the target type if successful, otherwise returns nullptr.

Syntax

dynamic_cast<target_type>(source_object)

Diagram: Dynamic Cast Example

```
Base* basePtr = new Derived();
Derived* derivedPtr = dynamic_cast(basePtr);
```

Week 16

Project Work

Project Work: Planning, Building, Testing, and Reviewing a Project

Welcome to this comprehensive guide on project work, covering essential stages from planning to deployment and review. We'll explore best practices for creating successful software projects using a library management system as an example. Prepare to gain insights into the lifecycle of software development.

Introduction to Project Management

What is Project Management?

Project management is the process of planning, organizing, and managing resources to achieve a specific goal.

Key Concepts

Scope, schedule, budget, resources, risk, communication, quality, and stakeholders are key concepts to consider.

Planning Phase: Requirements Gathering and Scope Definition

Understanding User Needs

Gathering detailed requirements through interviews, surveys, and workshops.

Defining Project Scope

Outlining the project's boundaries, deliverables, and milestones.

Creating a Project Plan

Developing a timeline, budget, and resource allocation plan.

Design Phase: Architectural Diagrams and UML Modeling

System Architecture

Defining high-level components and their interactions.

Database Design

Modeling data structures and relationships for efficient storage and retrieval.

User Interface Design

Creating wireframes and prototypes for user interaction.

Development Phase: Coding and Implementation in C++

1. Coding in C++: Choosing appropriate data structures, algorithms, and libraries.
2. Unit Testing: Writing tests for individual functions and modules.
3. Integration Testing: Testing the interaction between different components.

Testing Phase: Unit Testing, Integration Testing, and End-to-End Testing

1

Unit Testing

Testing individual units of code in isolation.

2

Integration Testing

Testing the interaction between different components.

3

End-to-End Testing

Testing the entire system from start to finish, simulating real-world scenarios.

Deployment Phase: Releasing the Application

1. **Build**

   Compiling and packaging the application.

2. **Deploy**

   Installing the application on the server.

3. **Test**

   Running tests in the production environment.

Maintenance Phase: Bug Fixes and Feature Updates

1. **Bug Fixes**

   Addressing reported bugs and defects.

2. **Feature Updates**

   Adding new features and functionality based on user needs.

3. **Security Patches**

   Implementing security updates and fixes.

Project Review: Lessons Learned and Continuous Improvement

1

Review Project Metrics

Analyze project performance, budget, and schedule.

2

Identify Lessons Learned

Document best practices and areas for improvement.

3

Continuously Improve

Apply lessons learned to future projects.

Diagram: Class Diagram for a Library Management System

```
+-----------------+
| Book            |
+-----------------+
| +isbn: string    |
| +title: string   |
| +author: string  |
| +availability: boolean |
+-----------------+

+-----------------+
| Member          |
+-----------------+
| +memberId: int   |
| +name: string    |
| +address: string |
+-----------------+

+-----------------+
| Loan            |
+-----------------+
| +loanId: int     |
| +memberId: int   |
| +isbn: string    |
| +dueDate: Date   |
+-----------------+
```

Week 16

Revision and Final Assessment

Revision and Final Assessment: A Comprehensive Review

This presentation will provide a comprehensive overview of key C++ topics, practical problem-solving techniques, and essential exam preparation strategies.

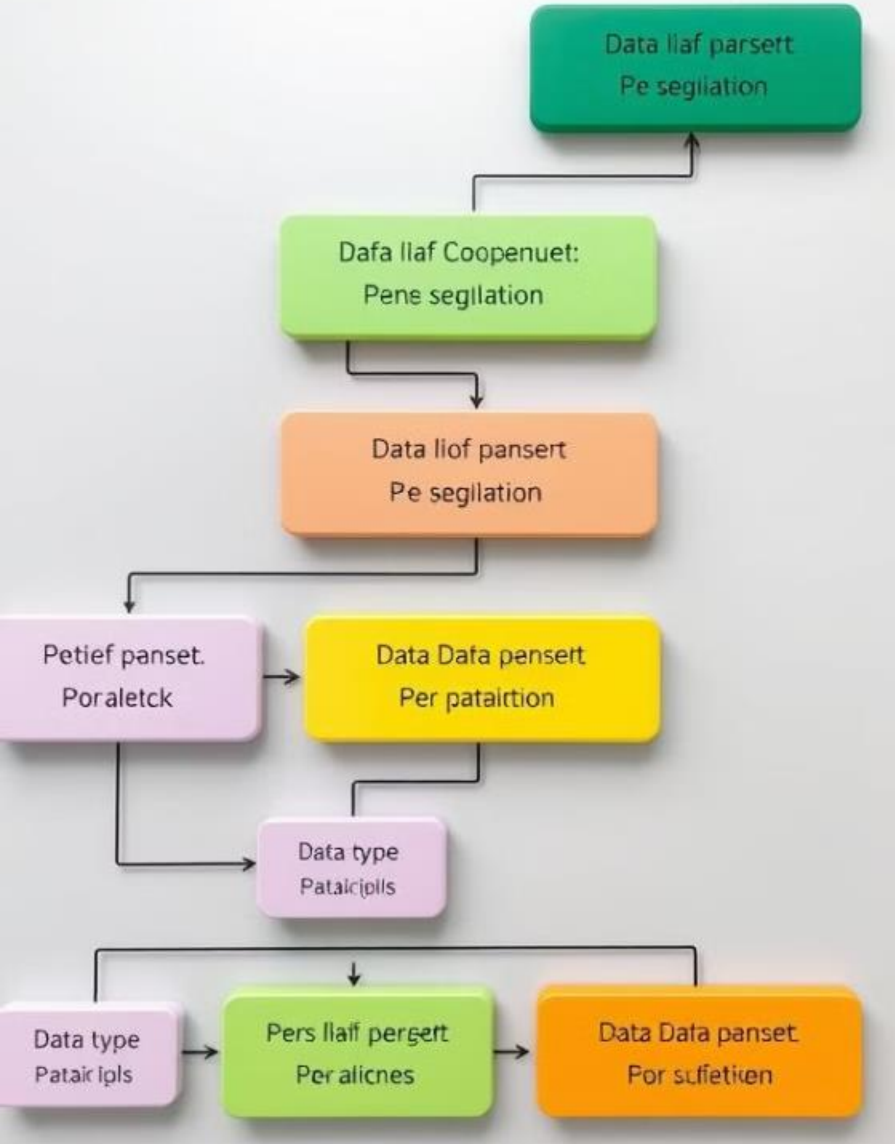
Introduction to C++ and its Core Features

History and Origins

C++ evolved from the C programming language. It was designed to be a powerful and versatile language for system programming and application development.

Key Features

Key features include object-oriented programming (OOP), generic programming, and memory management capabilities.



Data Types, Variables, and Operators

Basic Data Types

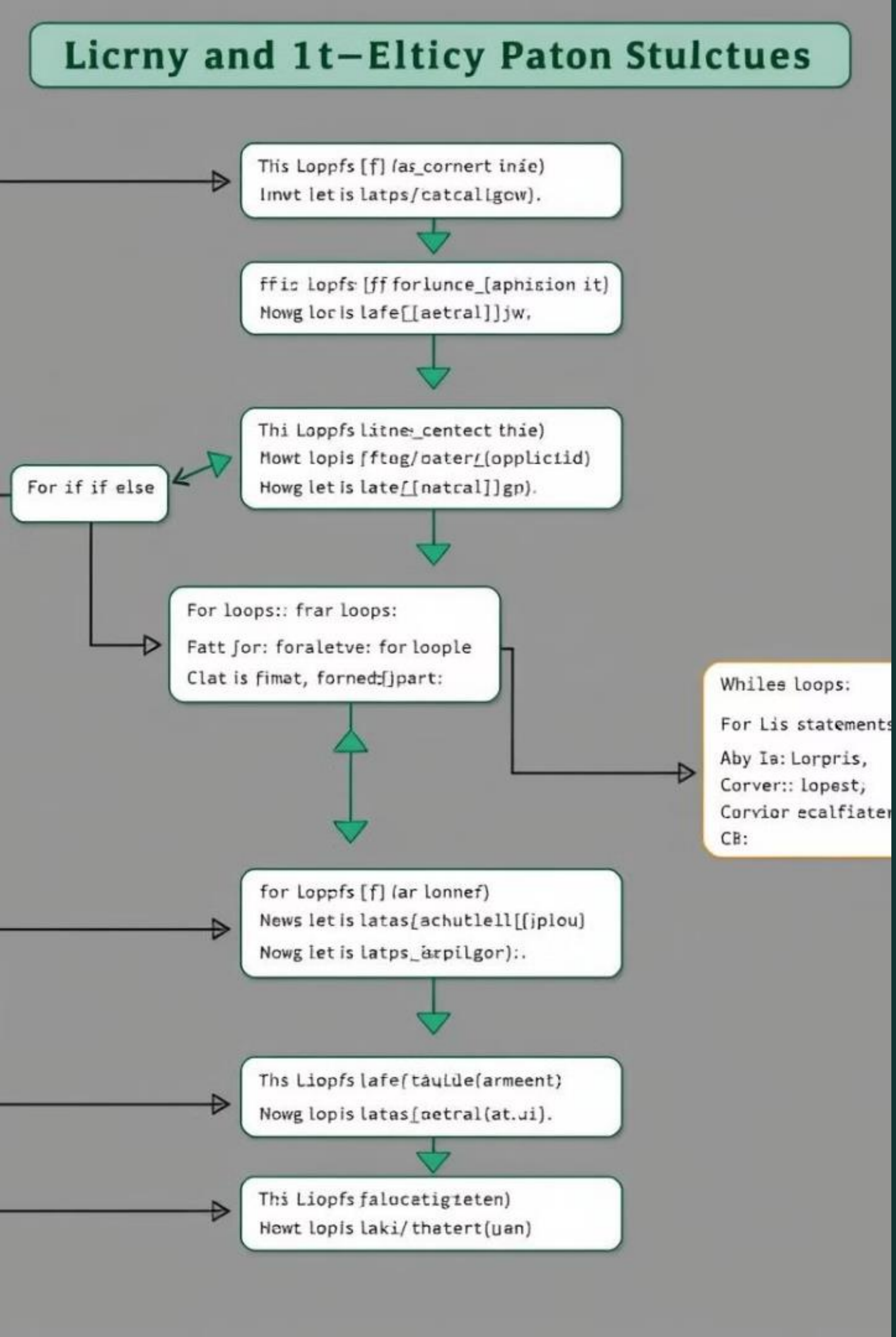
C++ supports various fundamental data types like int, float, char, bool, etc. Each type represents a specific kind of data.

Variable Declaration

Variables are used to store data in memory. They are declared using the data type followed by the variable name.

Operators

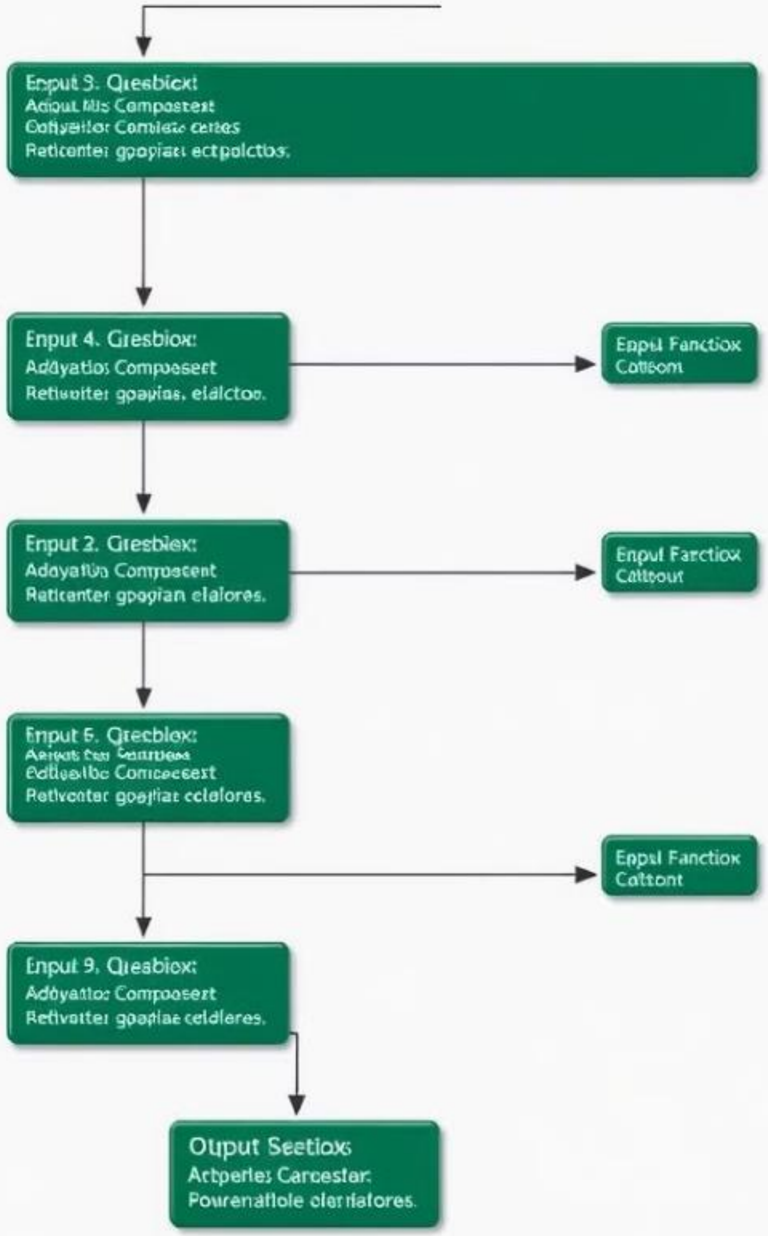
Operators are symbols that perform specific operations on variables and values. Examples include arithmetic, logical, and relational operators.



Control Structures: Conditional Statements and Loops

1 Conditional Statements: These allow the program to execute different code blocks based on specific conditions.

2 Loops: Loops repeatedly execute a block of code until a certain condition is met. This is useful for repetitive tasks.



Functions and Procedural Programming

1

Functions are reusable blocks of code that perform a specific task. They improve code organization and maintainability.

2

Procedural Programming: This paradigm focuses on breaking a program into a sequence of steps, with functions representing individual steps.

Arrays, Strings, and Pointers

Arrays

Arrays store collections of elements of the same data type. They are used to efficiently store and access related data.

Strings

Strings are sequences of characters. C++ provides built-in support for string manipulation, including concatenation and comparison.

Pointers

Pointers are variables that store memory addresses. They allow direct access to data stored in memory, enhancing performance and flexibility.

Object-Oriented Programming Concepts

1

Classes

Blueprints for creating objects, defining data members and member functions.

2

Objects

Instances of a class, containing data and behavior defined by the class.

3

Inheritance

Creating new classes based on existing ones, inheriting properties and behaviors.

4

Polymorphism

Allowing objects of different classes to be treated as objects of a common type.

5

Encapsulation

Bundling data and methods together, hiding implementation details.

File I/O and Exception Handling

1

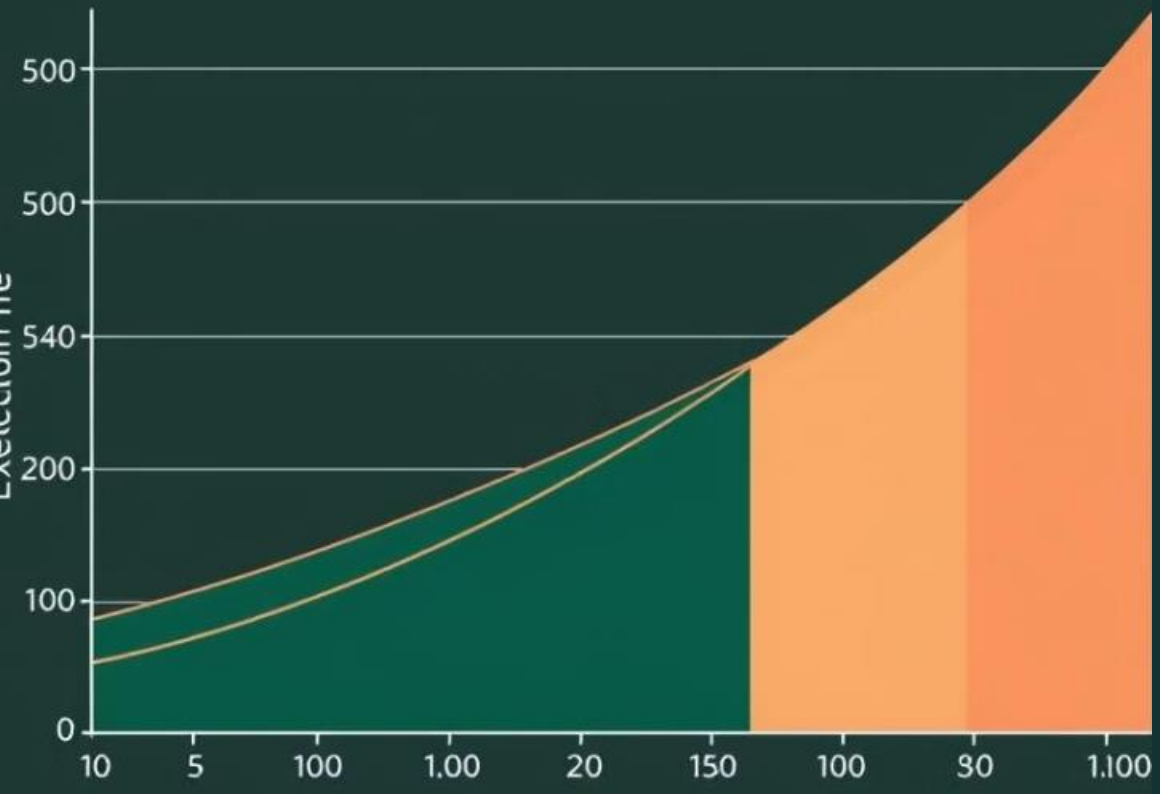
File I/O

Allows programs to interact with external files, reading data from them or writing data to them.

2

Exception Handling

Mechanisms for handling runtime errors and unexpected events, ensuring program stability and robustness.



Algorithm Analysis and Time Complexity

O(n)

Linear

Time complexity grows proportionally to the input size.

O(log n)

Logarithmic

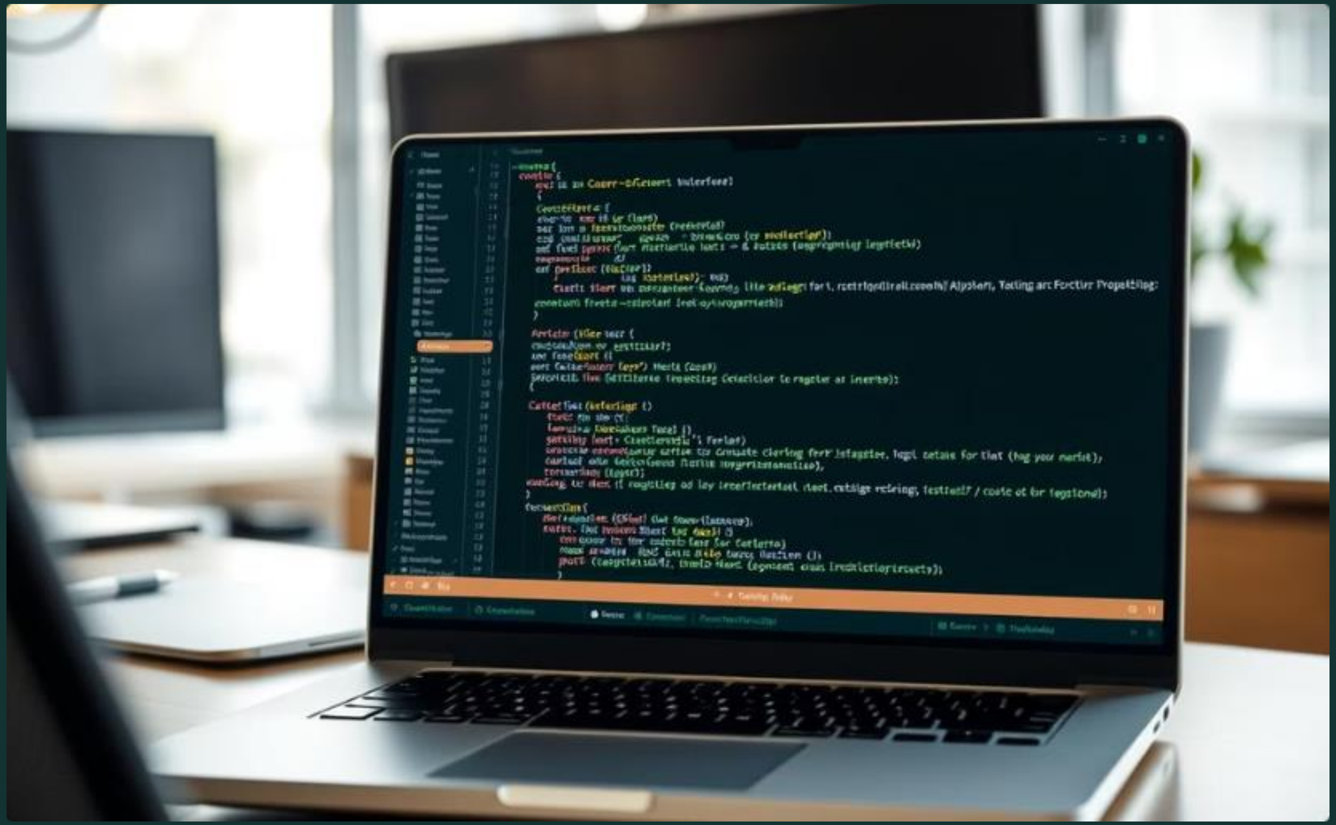
Time complexity increases slowly as input size increases.

O(n^2)

Quadratic

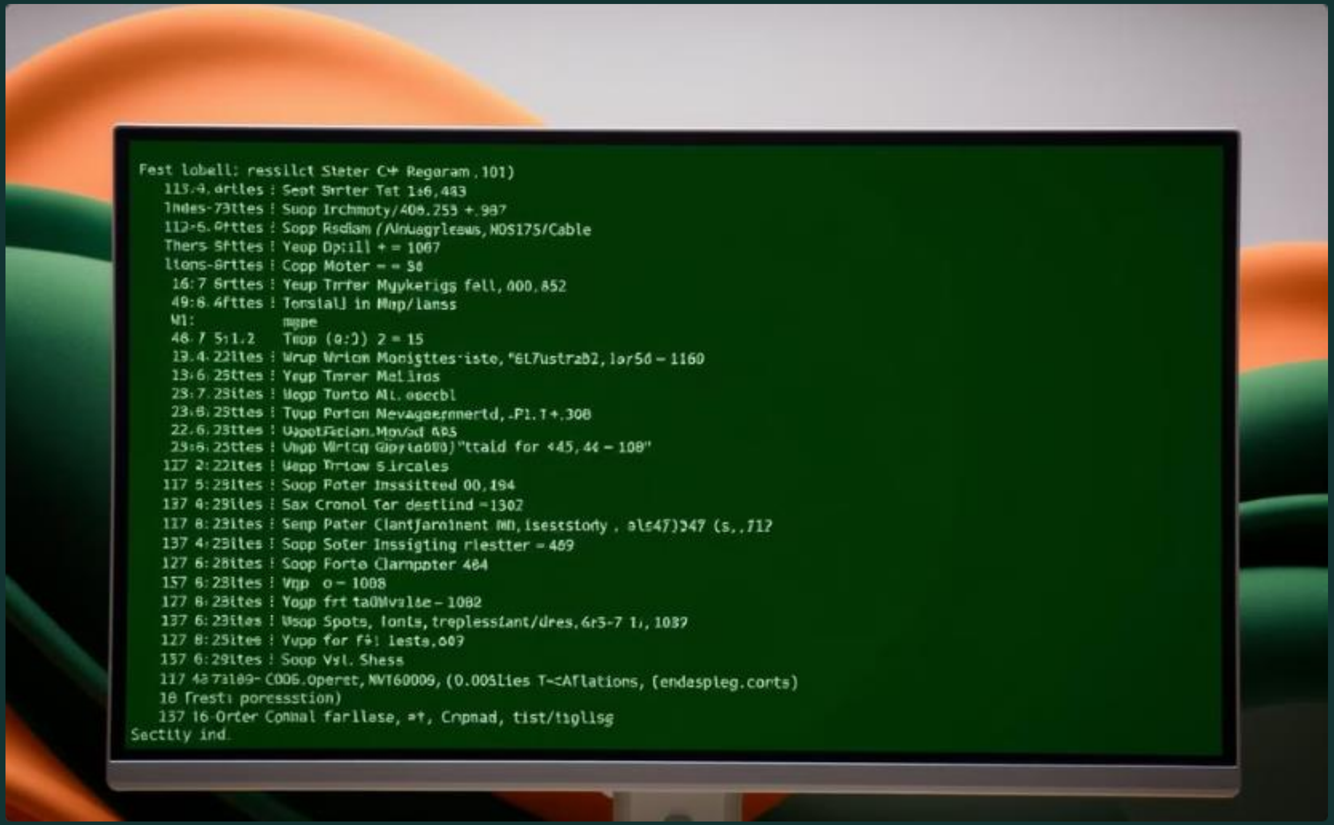
Time complexity grows quadratically with the input size.

Practical Problem Solving and Coding Exercises



Code Examples

Practical code examples illustrating solutions to common programming problems.



Output Results

Demonstration of program execution, displaying expected output and validating the solution.