ENPM702

Introductory Robot Programming

L8: Object-Oriented Programming - Part II

v1.0

Lecturer: Z. Kootbally

Semester/Year: Summer/2025

MARYLAND APPLIED
GRADUATE ENGINEERING

Table of Contents

Changelog

- *v1.0*: Original version.

Learning Objectives

By the end of this session, you will be able to:

- **Implement Modern C++ Features**
- **Understand `static` Members**
- **Understand the `this` Pointer**
- **Analyze Class Relationships**: Distinguish between **composition**, **aggregation**, and **inheritance** (including UML).
- **Implement Inheritance**: Apply single inheritance, understand access types, and properly initialize base class attributes.
- **Apply Polymorphism**: Differentiate compile-time from **runtime polymorphism**, using `virtual` methods and `override`.
- **Understand Abstraction**: Define abstraction and how it is achieved with **abstract classes** and **pure virtual functions**.
- **Differentiate Concrete Classes**: Understand their characteristics and relationship to abstract classes.
- **Use the `final` Keyword**: Prevent class inheritance and method overriding.

Visual Studio Code Tricks

To protect folders from accidental modification in VS Code, add the following in **.vscode/settings.json**

```
{
    "files.readonlyInclude": {
        "lecture1/**": true,
        "lecture2/**": true,
        "lecture3/**": true,
        "lecture4/**": true,
        "lecture5/**": true,
        "lecture6/**": true,
        "lecture7/**": true,
    }
}
```

The code snippet can be found in **vscode_scripts/lock_other_files.txt** (GitHub).

Code Enhancement

Let's elevate our `Robot` class using modern C++ features for better safety, performance, and maintainability.

Current Issues	Modern Solutions
Inefficient string handling	`std::string_view` parameters
Implicit conversions	`explicit` constructors
Ignored return values	`[[nodiscard]]` attributes
Unclear exception safety	`noexcept` specifications
Missing values not handled	`std::optional` for maybe values

What is `std::string_view`?

A lightweight, non-owning view into a string. It is essentially a pointer and size that lets you work with string data without copying or owning it.

A `std::string_view` contains just:

- A pointer to the beginning of the data.
- A size indicating how much data to view.

`#include <string_view>` to use `std::string_view`

Problem With Traditional Approaches

```cpp
// Using const std::string& - less flexible
void process_name(const std::string& name) { /* ... */ }

int main() {
    std::string john{"John Smith"};
    const char* jane{"Jane Doe"};

    process_name(john);      // OK
    process_name(jane);       // Creates temporary std::string -> expensive!
    process_name("Bob Johnson"); // Creates temporary std::string -> expensive!

    // Want substring? Need to create new string:
    process_name(john.substr(0, 4)); // Allocates memory for "John"
}
```

Solution With std::string_view

```cpp
void process_name(std::string_view name) { /* ... */ }

int main() {
    std::string john{"John Smith"};
    const char* jane{"Jane Doe"};

    process_name(john);      // OK - no conversion
    process_name(jane);      // OK - no conversion
    process_name("Bob Johnson"); // OK - no conversion

    // Substrings are free:
    process_name(john.substr(0, 4)); // No allocation - just adjusts pointer and
    ↪  size
}
```

Run the slide demo to compare the performance between `const std::string&` and `std::string_view`

Key Characteristics

- **Non-owning**: Doesn't manage memory.
- **Lightweight**: Usually 16 bytes (pointer + size).
- **Read-only**: Provides `const` access.
- **Efficient**: No copying or allocation.

Be Careful!

```cpp
std::string_view dangerous_function() {
    std::string temp{"temporary"};
    return temp;  // DANGER: returning view to destroyed object
}
```

```cpp
std::string_view safe_function() {
    static auto permanent{"permanent"};
    return permanent;  // OK: string literal has static storage
}
```

Use `std::string_view` when:

- Function only reads the string data.
- You want to accept any string-like input.
- Performance is critical.
- Working with substrings frequently.

Use `const std::string&` when:

- You specifically need `std::string` features not in `string_view`.
- The function is part of an older API that expects `std::string`.
- You need to store the reference long-term (be careful with `string_view` lifetime).

`std::string_view` is generally the better choice for read-only string parameters because it is more flexible, faster, and provides a cleaner interface while maintaining the same safety as `const std::string&` for temporary string objects.

What is `explicit`?

The `explicit` keyword prevents implicit conversions and copy-initialization for constructors and conversion operators, forcing users to be intentional about type conversions.

Problem Without explicit

```cpp
class Box {
public:
  Box(int size) : size_{size} {} // Non-explicit
  int get_size() const { return size_; }

private:
  int size_;
};
```

```cpp
//main.cpp
void ship_box(const Box &box) {
  std::cout << "Shipping box of size " << box.get_size() << '\n';
}

int main() {
  // These work, but intent is unclear:
  Box box1 = 10; // Looks like assignment, but creates Box
  ship_box(20);  // Looks like passing int, but creates Box
}
```

Solution Wit explicit

```cpp
class Box {
public:
  explicit Box(int size) : size_{size} {} // explicit
  int get_size() const { return size_; }

private:
  int size_;
};
```

```cpp
//main.cpp

int main() {
  // These cause compilation errors:
  //   Box box = 10;     // ERROR: no implicit conversion
  // ship_box(20);     // ERROR: no implicit conversion

  // Must be explicit:
    Box box{10};      // Clear: creating SafeBox object
    ship_box(Box{20}); // Clear: creating SafeBox object
}
```

When to use `explicit`?

- Single-parameter constructors (almost always).
- Multi-parameter constructors that might be confusing.
- Conversion operators that shouldn't happen implicitly.

Benefits

- Prevents bugs from accidental conversions.
- Makes code intent clearer.
- Improves API safety.
- Easier debugging (no hidden conversions).

What is `noexcept`?

The `noexcept` keyword specifies whether a function can throw exceptions. It serves as both a promise to the compiler and documentation for developers about exception safety.

Problem Without noexcept

```cpp
class Calculator {
public:
  int add(int a, int b) { return a + b; }
  int get_result() const { return result_; }
  bool is_valid() const { return valid_; }
private:
  int result_;
  bool valid_;
};
```

```cpp
//main.cpp
int main() {
  Calculator calc;
  // Compiler must prepare for potential exceptions:
  // - Exception handling overhead
  // - Stack unwinding preparation
  // - Reduced optimization opportunities
  int sum = calc.add(5, 10);  // Simple operation, but overhead added
}
```

Solution With noexcept

```cpp
class Calculator {
public:
  int add(int a, int b) noexcept { return a + b; }
  int get_result() const noexcept { return result_; }
  bool is_valid() const noexcept { return valid_; }
private:
  int result_;
  bool valid_;
};
```

```cpp
//main.cpp
int main() {
  Calculator calc;
  // Compiler can optimize aggressively:
  // - No exception handling overhead
  // - Better performance optimizations
  // - Clear contract: functions won't throw
  int sum = calc.add(5, 10);  // Optimized, no exception overhead
}
```

What the Compiler Must Do Without noexcept

- Generate Exception Handling Code

```cpp
int add(int a, int b) {  // Without noexcept
    return a + b;
}

// Compiler generates something like:
int add(int a, int b) {
    try {
        return a + b;
    } catch ( ... ) {
        // Stack unwinding code
        // Cleanup code
        // Rethrow mechanism
    }
}
```

What the Compiler Must Do Without noexcept

- **Preserve Stack Unwinding Information**

```cpp
void function_chain() {  // No noexcept
    int x{expensive_calculation()};
    int y{another_calculation()};
    return x + y;
}

// Compiler must keep track of:
// - What objects need destruction if exception thrown
// - How to unwind the stack properly
// - Exception propagation paths
```

What the Compiler Must Do Without noexcept

Conservative Register Usage

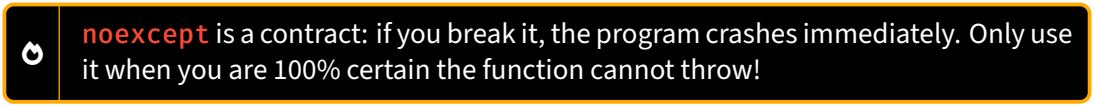
```cpp
int process_data(int a, int b, int c) {  // No noexcept
    int temp1{a * 2};
    int temp2{b * 3};
    int temp3{c * 4};
    return temp1 + temp2 + temp3;
}

// Without noexcept:
// - Must save/restore registers for exception safety
// - Cannot optimize away intermediate variables
// - Must maintain exception-safe state
```

When to use `noexcept`?

- Simple getters and setters.
- Move constructors and move assignment operators.
- Functions that genuinely cannot throw.

Benefits

- Better compiler optimizations.
- Improved STL container performance.
- Clear exception safety contracts.
- Self-documenting code.


noexcept is a contract: if you break it, the program crashes immediately. Only use it when you are 100% certain the function cannot throw!

Exception handling will be added to the appendix.

What is [[nodiscard]]?

The [[nodiscard]] attribute generates a compiler warning when the return value of a function is ignored. It communicates that ignoring the return value is likely a mistake.

Problem Without [[nodiscard]]

```cpp
class FileManager {
public:
  bool save_file(const std::string& filename) { /* save logic */ }
  std::string get_error_message() const { return error_msg_; }
  bool is_file_open() const { return file_open_; }
private:
  std::string error_msg_;
  bool file_open_;
};
```

```cpp
//main.cpp
int main() {
  FileManager fm;
  // These are probably bugs, but compiler won't warn:
  fm.save_file("data.txt");     // Did save succeed? Don't know!
  fm.get_error_message();       // Why call if not using result?
  fm.is_file_open();            // Forgot to check result!
}
```

Solution With [[nodiscard]]

```cpp
class FileManager {
public:
  [[nodiscard]] bool save_file(const std::string& filename) { /* save logic */ }
  [[nodiscard]] std::string get_error_message() const { return error_msg_; }
private:
  std::string error_msg_;
};
```

```cpp
//main.cpp
int main() {
  FileManager fm;
  // These now generate compiler warnings:
  fm.save_file("data.txt");     // Warning: ignoring return value
  fm.get_error_message();       // Warning: ignoring return value

  // Proper usage:
  if (fm.save_file("data.txt")) { /* handle success */ }
  auto error = fm.get_error_message(); // store result
}
```

When to use `[[nodiscard]]`?

- Accessors (return important data).
- Functions where ignoring result indicates a bug.
- Factory functions (prevent memory leaks).
- Status/validation functions.

Benefits

- Catches bugs at compile time.
- Self-documenting code (shows important returns).
- Forces proper error handling.
- No runtime cost (compile-time only).

Run the slide demo.

What is `std::optional`?

The `std::optional` (C++17) represents a value that may or may not exist. It provides type-safe way to handle operations that might fail, eliminating the need for "magic values" or error-prone null pointer patterns.

`#include <optional>` to use `std::optional`

Problem Without std::optional

```cpp
// Traditional error handling with magic values
[[nodiscard]] int divide(int a, int b) {
    if (b == 0) {
        return -1;  // Magic value for error - but -1 might be valid!
    }
    return a / b;
}
```

```cpp
//main.cpp
int main() {
    int result{divide(10, 0)};
    if (result == -1) { /* error */ }  // Fragile: -1 might be valid
}
```

Solution With std::optional

```cpp
[[nodiscard]] std::optional<int> divide(int a, int b) noexcept {
  if (b == 0) {
    return std::nullopt; // No value - clear error indication
  }
  return a / b; // Has value
}

int main() {
    std::optional<int> result{divide(10, 0)};
    if (result) {  // Clear: check if has value
        std::cout << *result << '\n';  // Safe access
        // or
        std::cout << result.value() << '\n';  // Safe access
    }
}
```

What is `std::nullopt`?

`std::nullopt` is a constant used with `std::optional` to represent an empty or uninitialized optional value.

Demonstration

```cpp
//main.cpp
int main() {
    std::optional<int> result{divide(10, 0)};

    // Different ways to check whether an std::optional contains a value
    if (result) {/*do something*/}
    if (result.has_value()) {/*do something*/}
    if (result != std::nullopt) {/*do something*/}
}
```

What is value_or?

`std::optional::value_or` is a convenient method that lets you retrieve the value inside an `std::optional` or provide a default if it's empty.

Demonstration

```cpp
int main() {
  std::optional<int> opt_with_value{10};
  std::optional<int> opt_empty{std::nullopt};

  std::cout << "opt_with_value: " << opt_with_value.value_or(-1) << "\n"; // 10
  std::cout << "opt_empty: " << opt_empty.value_or(-1) << "\n";           // -1
}
```

When to use `std::optional`?

- Functions that might fail (parsing, division by zero).
- Search operations (may not find target).
- Configuration values (may be unset).
- Database queries (may return no results).

Benefits

- **Type-safe**: compiler forces handling of “no value” case.
- **Expressive**: code clearly shows when something might be missing.
- **No magic values**: eliminates sentinel value confusion.
- **Exception safety**: `value_or()` provides safe defaults.

Core Enhancements

Feature	Purpose & Benefits
`std::string_view`	Lightweight, non-owning string views. Eliminates copies and temporary allocations for read-only parameters.
`explicit`	Prevents implicit conversions. Forces intentional construction, eliminating accidental bugs.
`noexcept`	Exception safety specification. Enables compiler optimizations and reduces overhead.
`[[nodiscard]]`	Compiler warnings for ignored returns. Catches bugs at compile time, self-documenting.
`std::optional`	Type-safe “maybe” values. Eliminates magic values and error-prone null patterns.

Key Benefits: Safety through compile-time checks, improved performance via optimizations, clearer code intent, and easier maintenance with fewer runtime errors.

`static` Attributes

A `static` attribute is a class-level variable shared among all instances of the class.

Key Properties

- **Shared State**: One copy shared by all objects.
- **Class Scope**: Exists even if no objects are created.
- **Lifetime**: Initialized once, exists until program ends.
- **Access**: Can be accessed via object or class name.

In C++17 and later, use `inline static` to define static variables directly inside the class. Older standards require an out-of-class definition in a .cpp file.

Traditional Static Variable Definition (Pre-C++17)

```cpp
// counter.hpp
class Counter {
public:
  static int count_; // Public static declaration
  Counter() { count_++; }

  static int get_secret() { return secret_; }

private:
  static int secret_; // Private static declaration
}; // class Counter
```

```cpp
// counter.cpp
int Counter::count_{0}; // definition
int Counter::secret_{42}; // definition
```

Inline Static Members (C_{17}^{++})

```cpp
// counter.hpp
class CounterInline {
public:
  inline static int count_{0}; // Public static
  CounterInline() { count_++; }

  static int get_secret() { return secret_; }

private:
  inline static int secret_{42}; // Private static
}; // class CounterInline
```

`inline static` (since C_{17}^{++}) eliminates the need for separate definitions in source files.

`static` Methods

`static` methods belong to the class, not any specific instance. They can be called without creating an object.

Key Properties

- **No Object Required**: Called using class name.
- **No Access to Non-Static Members**: No `this` pointer.
- **Common Use**: Utility functions and stateless helpers.

`static` methods are useful for operations that don't depend on object state.

`static` Methods

- `static` methods can access only `static` attributes.
- non-`static` methods can access both `static` and non-`static` attributes.

The `this` Pointer

The `this` pointer is an implicit parameter passed to all non-`static` member functions. It points to the object that invoked the function.

Key Properties

- Only exists in non-static member functions.
- Points to the calling object.
- Useful for disambiguating variable names and for fluent interfaces.

- In C++, `this` is an implicit pointer to the object.
- In Python, `self` is an explicit reference to the object.

Review **this_pointer_demo.hpp**

Class Relationships

Class relationships describe how classes interact and depend on each other. They represent real-world connections between objects and define the structure of your program.

Relationship	Strength	Description
Composition	Strongest	“Part-of” - contains objects that cannot exist without the container
Aggregation	Medium	“Has-a” - contains objects that can exist independently
Inheritance	Fundamental	“Is-a” - specialized class inherits from base class

What is Composition?

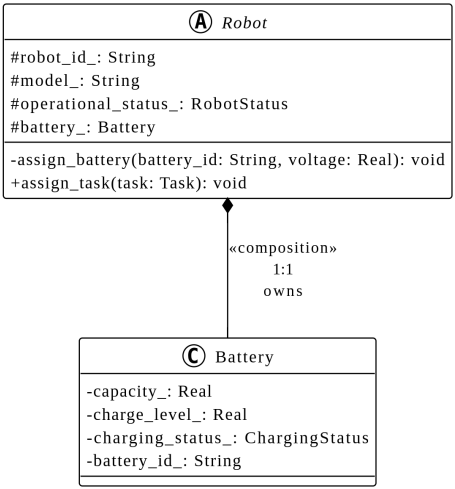
Composition is a “has-a” relationship where objects are composed of other objects as parts. The composed objects cannot exist independently of their container.

Example

```cpp
int main() {
    auto robot = std::make_shared<robotics::Robot>("WH-AMR-001", "AutoNav-Pro-X5",
    ↪  "BATX1", 200.0);
}
```

`Robot` owns and manages its `Battery` completely. The battery is an integral part of the robot and has no independent existence. If the robot is destroyed, the battery is also destroyed.

UML Representation



What is Aggregation?

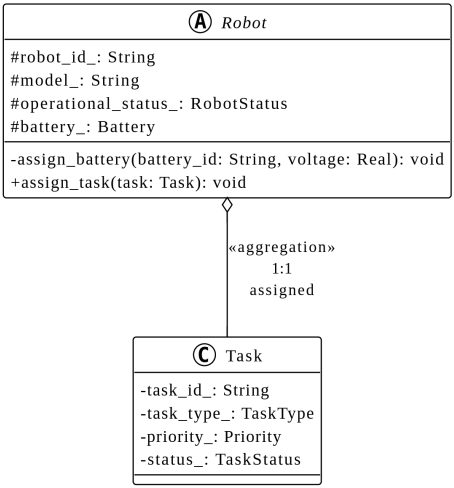
Aggregation is a “has-a” relationship where objects use or contain other objects as parts. The aggregated objects can exist independently of their container.

Example

```
int main() {
  auto robot =
      std::make_unique<robotics::Robot>("WH-AMR-001", "AutoNav-Pro-X5");

  auto maintenance_task = std::make_shared<robotics::Task>(
      "TSK-004", robotics::TaskType::MAINTENANCE, robotics::Priority::LOW);

      robot->assign_task(maintenance_task);
}
```

`Robot` can be assigned a `Task`. These two entities are independent. The same task can be assigned to a different robot.

UML Representation



What is Inheritance?

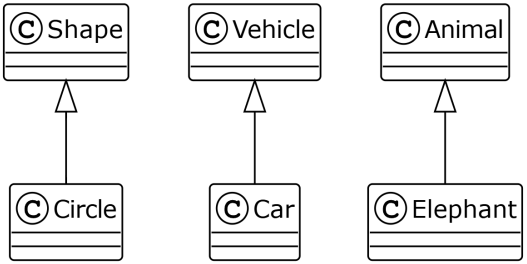
Inheritance is an “is-a” relationship where a specialized class (derived class) inherits its characteristics and behaviors from a more general class (base class). It enables polymorphism and code reuse.

Example

- `CarrierRobot` **is-a** `Robot`.
- `CarrierRobot` inherits common robot behaviors (move, status checking).
- `CarrierRobot` adds specialized behaviors (load/unload items).
- `CarrierRobot` overrides abstract methods with specific implementations.

`CarrierRobot` inherits all `public` and `protected` `Robot` capabilities but specializes in cargo transportation with additional load management features.

UML Representation



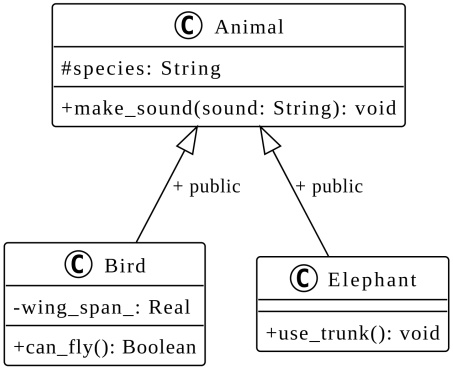
The protected Specifier

The protected specifier (# in UML) is especially useful in scenarios where you want derived classes to have access to certain attributes or methods of the base class, but you don't want those members to be public to the outside world. This maintains a degree of encapsulation while still providing flexibility for subclasses to use or modify certain internal details of the base class.

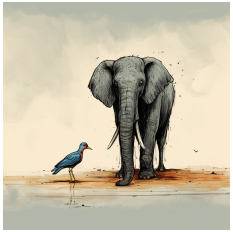
protected = private + accessible to derived classes.

Single Inheritance

Single inheritance: a class inherits from exactly one base class.

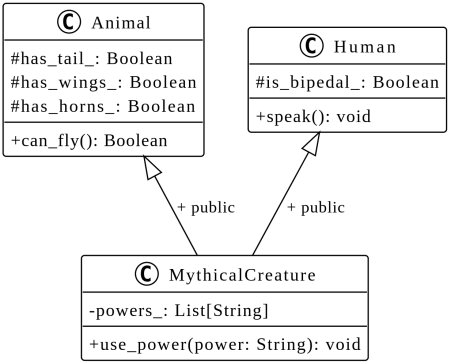


- `Bird` and `Elephant` inherit `Animal`'s `public` and `protected` members.
- UML diagrams typically don't show inherited members.



Multiple Inheritance

Multiple inheritance occurs when a class inherits from more than one base class. The derived class gains access to all `public` and `protected` members from all parent classes.



- `MythicalCreature` inherits all `protected` and `public` attributes from `Animal` and `Human`.

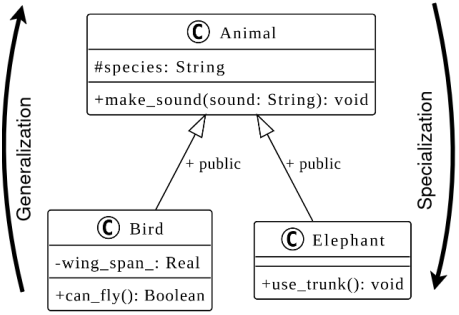
We focus exclusively on single inheritance in this course. For assignments and projects, you are welcome to use any inheritance approach.

Generalization

Bottom-up approach which should be used every time classes have specific differences and common similarities, so that the similarities can be grouped in a superclass and the differences maintained in subclasses.

Specialization

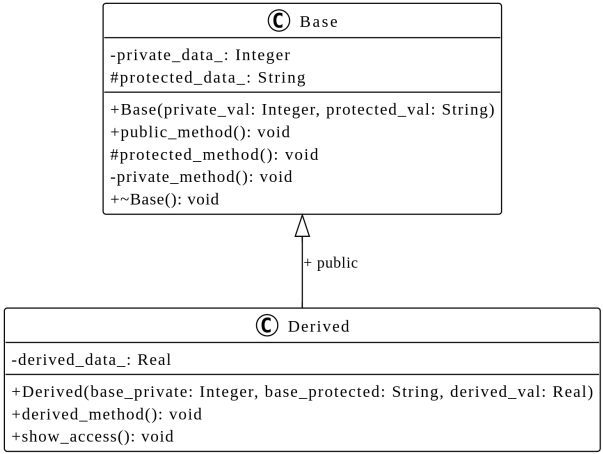
Top-down approach which creates new classes from an existing class. If certain properties only apply to some of the classes, subclasses can be created for these specific properties.



Inheritance Access Types

	Derived Class Member		
Base Class Member	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	not accessible (hidden)	not accessible (hidden)	not accessible (hidden)

- `private` members are inherited but hidden from derived classes.
- Default inheritance in C++ is `private`.



Review **inheritance_demo.hpp** and debug the code to observe how a derived class object contains both the base class components and its own attributes.


The constructors of a class must address the attributes specific to that class.

```cpp
// inheritance_constructor_demo.cpp
class Base {
 public:
  Base(int base_value = 50) :
    base_member_{base_value} {
        /*empty body*/
  }
 protected:
  int base_member_;
};  // class Base
//-----------------------------------
class Derived : public Base {
 public:
  Derived(double derived_value)
      : derived_member_{derived_value} {
        /*empty body*/
  }
 private:
  double derived_member_;
};  // class Derived
int main(){
    Derived derived(20.5);
}
```

- We previously saw that a derived class contains a portion of the base class.
- Before a derived class is initialized, the base class portion must be initialized.
- When `Derived derived(20.5)` executes, `Base(50)` is first executed: `base_member_` is initialized to `50`
- `Derived(20.5)` is then executed: `derived_member_` is initialized to `20.5`
- Control returns to `main()` and the program exits.
- With the way this program is written, `base_member_` is always initialized to the default value `50`.

Fix this code so that we can properly initialize `base_member_`

Approach #1

Add another parameter to the constructor for `Derived` and use it to initialize the attribute `base_member_` with *member initializer list*.

```
class Derived : public Base {
 public:
  Derived(double derived_value, int base_value)
      : derived_member_{derived_value}, base_member_{base_value} {
        /*empty body*/
  }
 private:
  double derived_member_;
};  // class Derived
int main(){
    Derived derived(20.5, 10);
}
```

C++ prevents classes from initializing inherited attributes in the member initializer list of a constructor.

Approach #2

Since we cannot use member initializer list, can we assigned `base_member_` in the body of the constructor?

```
class Derived : public Base {
 public:
  Derived(double derived_value, int base_value)
      : derived_member_{derived_value} {
         base_member_ = base_value;
  }
 private:
  double derived_member_;
};  // class Derived
int main(){
    Derived derived(20.5, 10);
}
```

Yes this works. However, we saw that this approach is performed in two steps and this will not work if the attribute is a `const` or a reference.

Approach #3

Explicit call of the base class constructor. As mentioned in slide 56, *constructors have to worry about their own attributes*.

```
class Derived : public Base {
 public:
  Derived(double derived_value, int base_value)
      : Base(base_value), derived_member_{derived_value} {
         /*empty body*/
  }
 private:
  double derived_member_;
};  // class Derived
int main(){
    Derived derived(20.5, 10);
}
```

- Add parameters for base class attributes in the derived class constructor.
- Explicitly call the base class constructor.

Polymorphism

Polymorphism (Greek: poly = many, morph = form) is a core OOP principle.

Polymorphism allows objects of different classes to be treated uniformly. C++ supports two types:

- Compile-time polymorphism
- Runtime polymorphism

Compile-Time Polymorphism

Compile-time polymorphism (also called *static polymorphism* or *early binding*) is resolved during compilation. The compiler determines which function to call at compile time.

Examples: *function overloading*, *operator overloading*, and *method redefinition*.

Method Order Check

```
// polymorphism_demo.cpp
int main() {
  Derived derived(20.5, 10);
  derived.test(); // Base::test() called
}
```

When a method is called on a derived class object:

1. The compiler first checks if the method exists in the derived class.
2. If not found, it searches up the inheritance hierarchy through base classes.
3. The compiler uses the first matching method it finds.

In the example, `Base::test()` is used since we did not provide a version for `Derived::test()`.

Method Redefinition

Method redefinition allows a derived class to provide its own implementation of a base class method when the base class version is too general.

Demonstration

```
// polymorphism_demo.cpp
int main() {
  Derived derived(20.5, 10);
  derived.test(); // Calls Derived::test(), not Base::test()
}
```

Method redefinition is compile-time polymorphism.

Runtime Polymorphism

Runtime polymorphism (*dynamic polymorphism* or *late binding*) determines which method to execute at runtime, not compile time.

Runtime polymorphism is achieved through `virtual` methods.

Warehouse Management System

Challenge

You need to command different robot types uniformly, but each performs tasks differently.

With References

```
void command_robot(robotics::CarrierRobot& robot) {robot.execute_task();}
void command_robot(robotics::SorterRobot& robot) {robot.execute_task();}
void command_robot(robotics::ScannerRobot& robot) {robot.execute_task();}

int main(){
    auto carrier_stack = robotics::CarrierRobot("CarrierX100", 5.0);
    command_robot(carrier_stack);

    auto sorter_stack = robotics::SorterRobot("SorterX100", 1.0);
    command_robot(sorter_stack);

    auto scanner_stack = robotics::ScannerRobot("ScannerX100", 5.0, 50.0);
    command_robot(scanner_stack);
}
```

With Pointers

```cpp
void command_robot(std::unique_ptr<robotics::CarrierRobot> robot) {
  robot->execute_task();
}
void command_robot(std::unique_ptr<robotics::SorterRobot> robot) {
  robot->execute_task();
}
void command_robot(std::unique_ptr<robotics::ScannerRobot> robot) {
  robot->execute_task();
}

int main(){
    auto carrier_pointer = std::make_unique<robotics::CarrierRobot>("CarrierX200", 5.0);
    command_robot(std::move(carrier_pointer));

    auto sorter_pointer = std::make_unique<robotics::SorterRobot>("SorterX200", 1.0);
    command_robot(std::move(sorter_pointer));

    auto scanner_pointer = std::make_unique<robotics::ScannerRobot>("ScannerX200", 5.0,
    ↪  50.0);
    command_robot(std::move(scanner_pointer));
}
```

Scalability & Redundancy

- There is a method for each derived class object. If we introduced a new subclass (e.g., `DroneRobot`) or if we remove an existing one, we need to modify our code.
- We had to overload the function `command_robot()` for each robot type.
  - Introducing the subclass `DroneRobot` to our project requires an additional overload of `command_robot()`.
  - Removing a robot class from our project requires us to eliminate one occurrence of `command_robot()`.
- The body of each function `command_robot()` is exactly the same, we have code duplication, which should be avoided in programming.

Runtime Polymorphism

Runtime polymorphism allows objects to be treated generically, with the specific behavior determined at runtime based on the actual object type.

Runtime polymorphism reduces code complexity, enables scalability, and eliminates redundancy.

```cpp
// One function works with any robot type!
void command_robot(std::unique_ptr<robotics::Robot> robot) {
    robot->execute_task();  // Calls correct derived class method
}

void command_robot(robotics::Robot& robot) {
    robot.execute_task();   // Calls correct derived class method
}
```

- Only one version of `command_robot()` needed for all robot types.
- The correct `execute_task()` method is called automatically at runtime.
- Works with `CarrierRobot`, `ScannerRobot`, `SorterRobot`, etc.
- Will work for any future `Robot` subclasses.

Requirements for Runtime Polymorphism

1. **Inheritance relationship** ✔
   - Derived classes inherit from a common base class.
2. **Base class pointers/references to derived objects**
   - Use `std::unique_ptr<Robot>`, `std::shared_ptr<Robot>`, or `Robot&` to refer to `CarrierRobot`, `ScannerRobot`, etc.
3. **Virtual methods in the base class**
   - Methods marked `virtual` enable dynamic dispatch.

With these three elements, the correct method is called at runtime based on the actual object type, not the pointer/reference type.

Base Class Pointers/References to Derived Objects

```cpp
// One function works with any robot type!
void command_robot(std::unique_ptr<robotics::Robot> robot) {
    robot->execute_task();  // Calls correct derived class method
}

void command_robot(robotics::Robot& robot) {
    robot.execute_task();   // Calls correct derived class method
}
```

Requirements for Runtime Polymorphism

1. **Inheritance relationship** ✔
   - Derived classes inherit from a common base class.
2. **Base class pointers/references to derived objects** ✔
   - Use `std::unique_ptr<Robot>`, `std::shared_ptr<Robot>`, or `Robot&` to refer to `CarrierRobot`, `ScannerRobot`, etc.
3. **Virtual methods in the base class**
   - Methods marked `virtual` enable dynamic dispatch.

With these three elements, the correct method is called at runtime based on the actual object type, not the pointer/reference type.

Virtual Methods in the Base Class

```
virtual void execute_task();
```

A `virtual` method is declared within a base class and can be overridden in a derived class. The primary purpose of `virtual` methods is to support dynamic polymorphism. By marking a method `virtual`, you indicate that the method can be overriden in derived classes and that the most-derived version of the method will be called, even when accessed through a base class pointer or reference to a derived object.

In other words:

- Without `virtual`, we *redefine* methods and we use compile-time polymorphism.
- With `virtual`, we override methods and we use runtime polymorphism.

Requirements for Runtime Polymorphism

1. **Inheritance relationship** ✔
   - Derived classes inherit from a common base class.
2. **Base class pointers/references to derived objects** ✔
   - Use `std::unique_ptr<Robot>`, `std::shared_ptr<Robot>`, or `Robot&` to refer to `CarrierRobot`, `ScannerRobot`, etc.
3. **Virtual methods in the base class** ✔
   - Methods marked `virtual` enable dynamic dispatch.

With these three elements, the correct method is called at runtime based on the actual object type, not the pointer/reference type.

Once a method is declared `virtual` in a base class, it remains `virtual` in all derived classes automatically. The method will use dynamic dispatch even without explicitly writing `virtual` in the derived classes.

Derived classes inherit virtual behavior automatically, making the `virtual` keyword optional. We include it explicitly in this course for clarity and best practice.

Mark the following derived class methods `virtual`:

- `robotics::CarrierRobot::execute_task()`
- `robotics::ScannerRobot::execute_task()`
- `robotics::SorterRobot::execute_task()`

The `override` Keyword

The `override` keyword explicitly marks methods that override virtual base class methods. While optional, it provides important benefits:

- **Clear Intent** – Makes it obvious that the method overrides a base class virtual method.
- **Compile-Time Safety** – The compiler verifies that:
  - The method actually overrides a virtual base class method.
  - The method signature exactly matches the base class.
  - Changes to base class signatures are caught at compile time, not runtime.

Always use `override` when overriding virtual methods to catch errors early and improve code clarity.

Use `override` on the following derived class methods:

- `robotics::CarrierRobot::execute_task()`
- `robotics::ScannerRobot::execute_task()`
- `robotics::SorterRobot::execute_task()`

Destructors

Destructors are special member functions that clean up resources when an object is destroyed. They are automatically called when an object goes out of scope or is explicitly deleted.

Key Characteristics

- **Naming**: Same as class name with `~` prefix (e.g., `~Robot()`)
- **No parameters**: Destructors cannot take arguments
- **No return type**: Not even `void`
- **Automatic call**: Called automatically when object lifetime ends
- **One per class**: Only one destructor allowed per class

Virtual Destructors and Inheritance

```cpp
class Robot {
public:
    virtual ~Robot() = default;  // Virtual destructor for base class
};

class CarrierRobot : public Robot {
public:
    ~CarrierRobot() override {   // Properly overrides base destructor
        // Cleanup carrier-specific resources
    }
};
```

Base classes used polymorphically must have `virtual` destructors to ensure proper cleanup of derived objects.

`= default` tells the compiler to generate the default implementation of a special member function (constructor, destructor, copy/move operations).

- **Explicit intent**: Shows you want the default behavior, not that you forgot to implement it.
- **Performance**: Compiler-generated functions are often more optimized.
- **Correctness**: Ensures proper implementation of complex operations like move constructors.

Abstraction

Abstraction is the principle of hiding complex implementation details while exposing only essential functionality through a simple interface.

Key Benefits

- **Simplicity**: Users interact with simple interfaces, not complex internals.
- **Modularity**: Implementation can change without affecting users.
- **Maintainability**: Easier to update and debug isolated components.

How is Abstraction Implemented in C++?

- **Abstract Classes**: Define interfaces without implementation details.
- **Pure Virtual Functions**: Force derived classes to provide specific implementations.
- **Encapsulation**: Hide internal data and expose controlled access.

```cpp
// Abstract interface - hides implementation complexity
class Robot {
public:
    virtual void execute_task() = 0;  // Pure virtual - must implement
    virtual ~Robot() = default;
protected:
    // Hidden complexity - users don't need to know these details
    bool validate_operation() const;
    void log_activity(std::string_view message) const;
};
```

Users call `robot->execute_task()` without knowing the complex internal operations each robot type performs.

Real-World Example: Coffee Machine

- **Simple Interface (What you see):**
  - Press “Espresso” button.
  - Press “Macchiato” button.
  - Press “Clean” button.
- **Hidden Complexity (What you don’t see):**
  - Water temperature control.
  - Valve operations.
  - Pressure regulation.
  - Circuit board operations.
  - Grinding mechanisms.

If the manufacturer changes the internal circuit board, you still press the same buttons. The interface remains constant while implementation changes.

Concrete Classes

A concrete class is a class that can be instantiated.

Key Characteristics

- **Complete Implementation**: All methods have implementations (no pure virtual functions).
- **Instantiable**: You can create objects directly from concrete classes.
- **Derived from Abstract**: If inheriting from abstract classes, must implement all pure virtual functions.

Example

```cpp
// Abstract base class
class Robot {
public:
    virtual void execute_task() = 0;  // Pure virtual
};

// Concrete derived class
class CarrierRobot : public Robot {
public:
    void execute_task() override {    // Implementation provided
        // Carrier-specific task execution
    }
};

// Usage
CarrierRobot carrier("ROBOT001", 50.0);  // Can instantiate
// Robot robot;  // ERROR: Cannot instantiate abstract class
```

The `final` Keyword

The `final` keyword prevents inheritance and method overriding.

- **Prevent Class Inheritance**

```
class CarrierRobot final : public Robot {
    // Implementation
};
// class SpecialCarrier : public CarrierRobot {};  // ERROR: Cannot inherit
```

- **Prevent Method Overriding**

```
class Robot {
public:
    virtual void execute_task() = 0;
    virtual void move() final {  // Cannot be overridden
        // Final implementation
    }
};

class CarrierRobot : public Robot {
    void execute_task() override { /* OK */ }
    // void move() override { }  // ERROR: Cannot override final method
};
```

Next Class

- Lecture 9: ROS (Part 1).
  - Quiz on OOP.