ENPM702

Introductory Robot Programming

L7: Object-Oriented Programming - Part I

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:

- Understand the fundamentals of object-oriented programming and its four core principles: encapsulation, inheritance, polymorphism, and abstraction.
- Design and implement classes using modern C++17 features including `std::string_view`, `std::optional`, inline static variables, and `[[nodiscard]]` attributes.
- Apply RAII principles and smart pointer management (`std::unique_ptr`, `std::shared_ptr`) for automatic resource management and memory safety.
- Implement modern constructor patterns including delegating constructors, member initializer lists, and the `explicit` keyword for type safety.
- Use advanced C++17 features such as Class Template Argument Deduction (CTAD), structured bindings, and efficient string handling with `std::string_view`.
- Follow modern C++ best practices including the Rule of Zero, const-correctness, `noexcept` specifications, and factory patterns for robust, maintainable code.

Modern C++17 Philosophy

Modern C++17 introduces powerful features that make object-oriented programming more expressive, safer, and efficient. The evolution emphasizes **type safety**, **resource management**, and **performance** while maintaining backward compatibility.

C++17 features like `std::optional`, `std::string_view`, Class Template Argument Deduction (CTAD), and structured bindings transform how we write object-oriented code, making it more readable and less error-prone.

C++17 Features for OOP

- `std::optional` – Safe error handling without exceptions.
- `std::string_view` – Efficient string parameter passing.
- **Class Template Argument Deduction (CTAD)** – Reduced template boilerplate.
- **Structured Bindings** – Elegant multiple return values.
- `[[nodiscard]]` – Compiler warnings for ignored return values.
- **Inline Static Variables** – In-class static member initialization.
- **Enhanced Aggregate Initialization** – More flexible object construction.

Modern C++ Principles

DO:

- ☐ RAII (Resource Acquisition Is Initialization).
- ☐ Prefer stack allocation over heap.
- ☐ Use smart pointers for dynamic allocation.
- ☐ Embrace `const` correctness.
- ☐ Use `noexcept` where appropriate.
- ☐ Prefer uniform initialization.

AVOID:

- ☐ Raw pointers for ownership.
- ☐ Manual memory management.
- ☐ Implicit conversions.
- ☐ Non-const methods when not modifying.
- ☐ Bare `new` and `delete`.

Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that organizes code around objects rather than functions. It emphasizes creating reusable, modular code through the combination of data and the methods that operate on that data.

The Four Pillars of OOP:

- Encapsulation – Bundling data with methods and controlling access.
- Inheritance – Creating new classes from existing ones.
- Polymorphism – Objects sharing behaviors through common interfaces.
- Abstraction – Hiding implementation details and exposing necessary features.

Advantages

- **Modularity** – Break complex problems into manageable pieces.
- **Reusability** – Classes can be reused across projects through inheritance.
- **Flexibility** – Polymorphism enables dynamic behavior.
- **Maintainability** – Changes are localized to specific classes.

Disadvantages

- **Learning curve** – Requires understanding of OOP concepts.
- **Design overhead** – More upfront planning required.
- **Program size** – Can be larger than procedural code.
- **Performance** – Small overhead from virtual functions.

OOP is not a solution for everything. It is not suitable for all types of problems, and not everything decomposes naturally into classes.

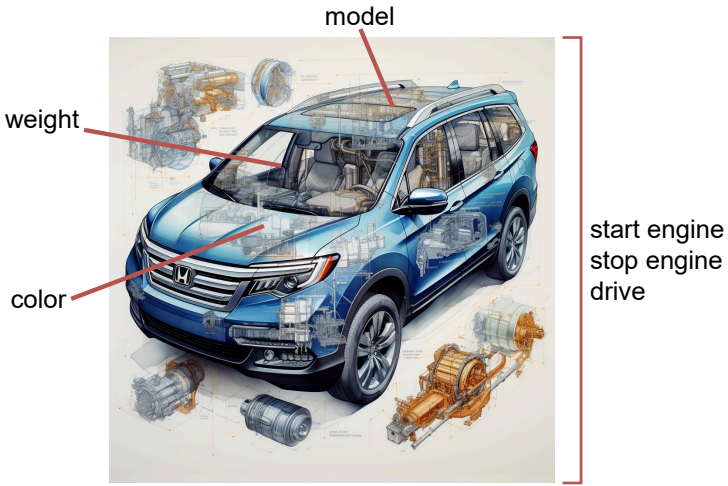
Classes and Objects

Class

A **class** is a blueprint which defines the **attributes** (data members) and the **methods** (member functions) that can operate on the attributes.

Object

An **object** is an instance of a class. Instantiating a class means creating an object of the class.



Class `Vehicle` with attributes and methods.

Characteristics

Caption

- Multiple objects can be created from the same blueprint.

Characteristics

- Each object is unique. For example, two red vehicles are considered two different objects; if one of these vehicles is sold, the other remains unaffected.
  - This means that each object has its own set of attributes (e.g., `color`, `weight`, `width`).

Characteristics

- All objects share the same methods. For example, `start_engine()` is used by all vehicles.
  - A black vehicle (`black_vehicle`) and a red vehicle (`red_vehicle`), both created from the same class, use the **identical** `start_engine()` method implementation. Each object does **NOT** have its own copy of the method code.
  - When `black_vehicle.start_engine()` is called, the method operates on `black_vehicle`'s specific attributes (like its `speed` value). Similarly, `red_vehicle.start_engine()` operates on `red_vehicle`'s attributes.
  - The `this` pointer ensures that each method call automatically accesses the correct object's data members.

- Multiple Instantiation: A single class blueprint can create numerous different objects, each with unique identities and memory locations.
- Object Independence: Each object maintains its own set of attributes and state. Even objects with identical attribute values (e.g., two red vehicles) are completely separate entities that don't affect each other.
- Shared Method Implementation: All objects from the same class use identical method code stored once in memory. There are no duplicate copies of methods for individual objects.

```cpp
#include <string>
#include <string_view>
#include <optional>

class Vehicle {
private:
    std::string model_;
    std::string color_;
    double weight_;
    bool is_running_ = false;  // C++17: in-class initialization

public:
    // Modern constructor with string_view
    Vehicle(std::string_view model, std::string_view color, double weight)
        : model_{model}, color_{color}, weight_{weight} {}

    // Modern const-correct accessors with [[nodiscard]]
    [[nodiscard]] std::string_view get_model() const noexcept { return model_; }
    [[nodiscard]] std::string_view get_color() const noexcept { return color_; }
    [[nodiscard]] double get_weight() const noexcept { return weight_; }

    // Methods
    void start_engine() noexcept { is_running_ = true; }
    void stop_engine() noexcept { is_running_ = false; }
    [[nodiscard]] bool is_running() const noexcept { return is_running_; }
}; // class Vehicle
```

- Class Structure & Syntax: Use `class` keyword to define a class. Class definition ends with a semicolon `;`. End-of-class comment.
- Naming Conventions: Class names use PascalCase (`Vehicle`). Attributes use snake_case with trailing underscore. Methods use snake_case.
- Access Specifiers: `private`, `public`, or `protected`.
- Modern C++17 Features: In-class initialization: `bool is_running_ = false;`. `std::string_view` for constructor parameters (avoids string copying). `[[nodiscard]]` attribute prevents ignoring important return values. `noexcept` specification for exception safety guarantees.
- Constructor Design: Member initializer list `: model_{model}, color_{color}, weight_{weight}`. Uniform initialization with braces. Parameters use `std::string_view` for efficiency.
- Method Categories: Accessors/Getters: `get_model()`, `get_color()`, `get_weight()`. Mutators: `start_engine()`, `stop_engine()`.
- Best Practices Demonstrated: Const-correctness: `const` methods for non-modifying operations. Encapsulation: Private data with controlled public access. Efficient return types: `std::string_view` instead of `std::string` copies. Clear method names: Self-documenting function names.

```cpp
#include <memory>
#include <iostream>

int main() {
    // Stack allocation (preferred)
    Vehicle honda_civic{"Honda Civic", "Blue", 1200.5};

    // Dynamic allocation with smart pointers
    auto ford_f_150 = std::make_unique<Vehicle>("Ford F-150", "Red", 2500.0);
    auto tesla_model_3 = std::make_shared<Vehicle>("Tesla Model 3", "White", 1600.0);

    // Using objects
    honda_civic.start_engine();
    std::cout << honda_civic.get_model() << " is "
              << (honda_civic.is_running() ? "running" : "stopped") << '\n';

    // Accessing through smart pointers
    ford_f_150->start_engine();
    std::cout << ford_f_150->get_model() << " weight: " << ford_f_150->get_weight() <<
    ↪  "kg\n";

    tesla_model_3->start_engine();
    tesla_model_3->end_engine();
}
```

Design Phase

The design phase in OOP is crucial as it establishes the blueprint for the application. It involves transforming the requirements into a workable structure that serves as a foundation for further software development.

- The Requirement Analysis clearly describes what the system is supposed to achieve.
- The Modeling phase represents and visualizes the system's structure and behavior through diagrams.

Requirement Analysis

What is Requirements Analysis?

Defines **what** the system must do before designing **how** it will work.

1. Functional Requirements - Define what the system must do and which specific capabilities it must provide.
2. Non-Functional Requirements - Specify how well the system must perform in terms of quality attributes like performance, reliability, and safety.
3. Technical Constraints - Establish design limitations and architectural principles that restrict implementation choices.
4. Success Criteria - Set measurable outcomes that determine whether the project has achieved its goals.

Open **requirement_analysis_warehouse_robots.pdf**

Modeling Phase

What is System Modeling?

Defines **how** the system will be structured and designed to meet the requirements.

1. Class Design - Define system structure using classes, attributes, methods, and relationships to represent real-world entities.
2. Sequence Diagrams - Model dynamic behavior by showing how objects interact over time to accomplish specific tasks.
3. System Architecture - Establish overall system organization including inheritance hierarchies, composition, and aggregation relationships.
4. Implementation Guidelines - Provide development phases, testing strategies, and deployment considerations for building the system.

Open modeling_warehouse_robots.pdf

Modeling Languages

There are many different types of graphical and textual modeling languages that help developers in the design and implementation of computer objects, systems, and architectures.

Two of these graphical modeling languages are known as UML and SysML.

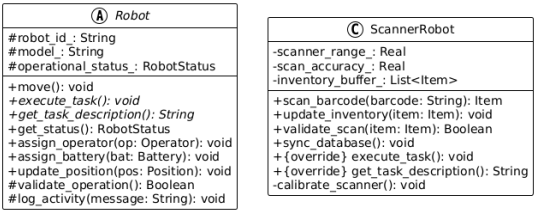
- UML is mainly used by software engineers to visualize software systems and consists of 14 diagrams.
- SysML is an extension of UML designed for use in systems engineering applications and consists of 9 diagrams.

In this course, we will use one type of **structural diagram** (class diagram) and one type of **behavioral diagram** (sequence diagram) to describe the domain.

- A structural diagram lets you visualize the static parts that need to be present in a software system. This type of diagram is often used to document software architecture. Examples of structural diagrams include component diagrams and class diagrams.
- A behavioral diagram lets you visualize the things that happen within the software. This type of diagram helps you to document a software system's functionality. Examples include sequence diagrams and use case diagrams.

Class Diagram

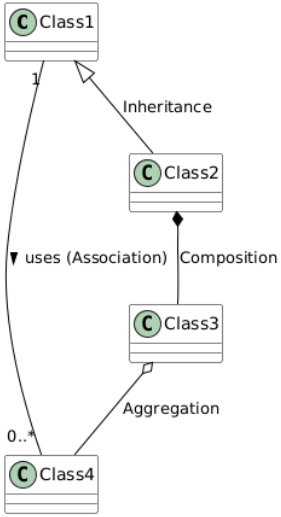
- A class diagram is a type of UML structural diagram that represents the static structure of a system by showing its classes, their attributes, methods, and the relationships among the classes. Class diagrams are widely used in object-oriented programming to design and visualize the architecture of software systems.



Classes

A class is depicted as a rectangle divided into three sections:

- **Class Name**: The top section contains the name of the class.
- **Attributes (member data)**: The middle section lists the properties or fields (also known as attributes) of the class.
- **Methods (member functions)**: The bottom section lists the methods or functions that the class provides.
- The types are generic.
- Visibility indicators:
  - + (`public`)
  - - (`private`)
  - # (`protected`)



Relationships

A class diagram describes the relationships between classes. Some relationship descriptions can be enhanced with a label and a direction.

- **Association** represents a general relationship between two classes.
- **Aggregation**: A specialized form of association indicating a **whole-part** relationship where the part can exist independently of the whole.
- **Composition**: A stronger form of aggregation where the part cannot exist independently of the whole.
- **Inheritance**: Indicates that one class (subclass) inherits attributes and methods from another class (superclass).

Multiplicities

Multiplicities define how instances of one class can be associated with instances of another class. It is often shown on the association lines between classes.

- **1** (exactly one), **0..1** (zero or one), **1..*** (at least one), * or **0..*** (zero or more).

Sequence Diagram

A UML sequence diagram is a type of behavioral diagram that illustrates the dynamic interactions between objects or participants in a system over time. It shows the chronological order of message exchanges between different objects, depicting how they collaborate to accomplish a specific task or scenario, and provides a visual representation of the temporal flow of system behavior.

Implementation Phase

The implementation phase comes after the design phase. It consists of using the diagrams to write the code for the application.

Some tools can generate C++ code from UML class diagrams (e.g., Visual Paradigm and starUML).

- lecture7
  - src
    - battery.cpp
    - carrier_robot.cpp
    - inventory_database.cpp
    - item.cpp
    - main.cpp
    - operator.cpp
    - position.cpp
    - robot.cpp
    - scanner_robot.cpp
    - sorter_robot.cpp
    - task.cpp
  - include
    - battery.hpp
    - carrier_robot.hpp
    - inventory_database.hpp
    - item.hpp
    - operator.hpp
    - position.hpp
    - robot.hpp
    - scanner_robot.hpp
    - sorter_robot.hpp
    - support.hpp
    - task.hpp

Modern Practices

File Inclusions

```
// angle brackets for standard
    library
#include <iostream>
#include <memory>
#include <string>
#include <string_view>
#include <optional>

// quotes for project headers
#include "robot.hpp"
```

Header Protection (modern approach)

```
#pragma once

#include <string>
// other includes

namespace robotics {
    class Robot {
        // class definition
    };
}
```

As a rule of thumb: 1) header files are included but not compiled, and 2) source files are compiled but not included.

ToDo

1. `Robot` class **declaration** in **robot.hpp**
2. `Robot` class **implementation** in **robot.cpp**
3. `Robot` class **instantiation** in **main.cpp**
   - Access class members with the dot operator (for stack objects).
   - Access class members using the arrow operator (for pointers).
4. Edit **CMakeLists.txt** and run.
5. Check the structure of the created objects with the debugger.

When you use `std::make_unique<T>()` or `std::make_shared<T>()` without any arguments, it performs value initialization of the object `T`. This behavior ensures that:

- **Fundamental types** are zero-initialized.
- **Class types** are initialized via the default constructor if it exists; otherwise, members are zero-initialized.

Access Specifiers

Access specifiers determine the accessibility of the members (attributes and methods) of a `class` or `struct`. They play a crucial role in **encapsulation**, a fundamental concept in object-oriented programming. There are three primary access specifiers in C++.

- `public` – Members declared under the `public` specifier can be accessed from outside the class and by derived classes.
- `private` – Members declared as `private` are restricted to the class in which they are declared. They can't be accessed from outside the class or by derived classes.
- `protected` – `protected` members are somewhere between `public` and `private`. They can't be accessed from outside the class, but they can be accessed by derived classes. We see this in detail in the **inheritance** section.

If you do not provide any access specifier, C++ will make all your members `private`.

Encapsulation

Encapsulation (or data hiding) is one of the principles of OOP. The main purpose of encapsulation is to provide security to the data by restricting its access to the public (the end users).

- In C++, encapsulation is performed via the access specifier `private`.
- As a developer, you **must encapsulate** all attributes to protect them.
- If you want the user to access and/or modify some attributes, then provide **accessors (getters)** and **mutators (setters)**.
- The C++ Standard Library encapsulates all their attributes. This is why you can't access them directly.

ToDo

1. Make all attributes of the `Robot` class `private`.
2. Make methods of the `Robot` class `public` only if you intend to use them outside the class; otherwise, keep them `private` (or `protected`).
3. Re-run the `main()` function.

Accessors (Getters)

An accessor (or getter) is a `public` method that allows the user to access `private` and `protected` attributes.

Key Points

- The return type of the accessor is the type of the attribute you are retrieving.
- Accessors do not need to have parameters.
- Accessors should provide **read-only** access to data. Therefore, they should return by value or by `const` reference (never by non-`const` reference).
- Accessors should be declared as `const`.
- Accessors are **usually** written directly in the class definition.

```cpp
// return by value
robotics::RobotStatus robotics::Robot::get_status() const{
    return operational_status_;
}
```

or

```cpp
//  return by const reference
const robotics::RobotStatus& robotics::Robot::get_status() const{
    return operational_status_;
}
```

`const`-Correctness with Accessors

When a method is declared with `const` at the end, it creates a contract that guarantees the function will not modify the state of the object. This means no member variables can be changed when this function is called.

Demonstration

```
robotics::RobotStatus robotics::Robot::get_status() {
    operational_status_ = robotics::RobotStatus::ACTIVE; // Oof!
    return operational_status_;
}
```

```
robotics::RobotStatus robotics::Robot::get_status() const{
    operational_status_ = robotics::RobotStatus::ACTIVE; // Error
    return operational_status_;
}
```

- Key Rule #1: `const` Methods Are Universally Callable.
  - A `const` method can be called on both `const` and non-`const` instances of the class.
- Key Rule #2: Non-`const` Methods Have Restrictions.
  - If a method is not marked as `const`, it cannot be called on an object that is declared `const` because the compiler cannot guarantee that the function will not modify the object's state.

Compiler's Perspective

- `const` methods: "I promise not to modify the object" → Safe to call anytime.
- Non-`const` methods: "I might modify the object" → Only safe on modifiable objects.

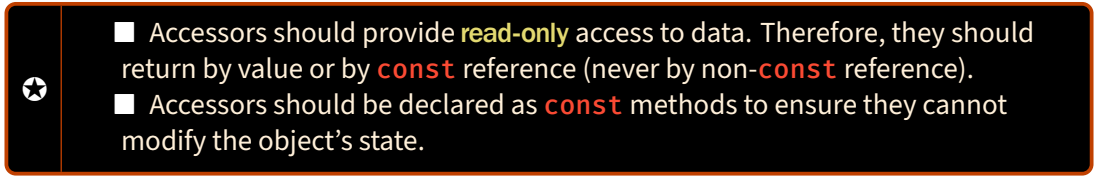
Method Type	Can Call On `const` Object?	Can Call On Non-`const` Object?
`const` method	✔ Yes	✔ Yes
Non-`const` method	✘ No	✔ Yes

Demonstration

```cpp
// robot.hpp
robotics::RobotStatus get_status() const;   // const
std::string get_model();                    // non-const
```

```cpp
// main.cpp
const robotics::Robot robot1;               // robot1 is const
std::cout << robot1.get_status() << '\n';   // OK: Key Rule #1
std::cout << robot1.get_model() << '\n';    // Error: Key Rule #2

robotics::Robot robot2;                     // robot2 is non-const
std::cout << robot2.get_status() << '\n';   // OK: Key Rule #1
std::cout << robot2.get_model() << '\n';    // OK
```


Accessors should provide read-only access to data. Therefore, they should return by value or by const reference (never by non-const reference).
Accessors should be declared as const methods to ensure they cannot modify the object's state.

ToDo

Write the accessors for the `Robot` class attributes that you believe may require **access** from outside the class.

Mutators (Setters)

A mutator (or setter) is a `public` method that allows the user to modify `private` and `protected` attributes of a class.

Key Points

- Mutators typically have a `void` return type, indicating they do not return any value.
- Mutators take one parameter, which is the value that will be assigned to the attribute.
- The parameter can be passed:
  - By value, where a copy of the value is made (suitable for small data types).
  - By `const` reference, which avoids unnecessary copying, especially for large objects like strings.
- Parameters should not be passed by non-`const` reference, as it would allow modification of the caller's argument.
- Mutators are commonly implemented directly within the class definition.

```
// pass by value
void robotics::Robot::set_model(std::string model) {model_ = model;}
// pass by const ref
void robotics::Robot::set_model(const std::string& model) {model_ = model;}
```

ToDo

Write the mutators for the `Robot` class attributes that should be modified outside the class.

Constructors

A constructor is a special `public` method of a class that is automatically invoked whenever an object of that class is created.

The primary goals of a constructor are:

- Initialization – One of the main purposes of a constructor is to initialize an object's attributes when the object is created. This ensures that the object starts its life in a consistent and expected state.
- Resource Allocation – If an object needs to acquire certain resources (like dynamic memory) when it is created, the acquisition can be done in the constructor.

Characteristics

- **Name Matching the Class** – A constructor has the same name as its class.
- **No Return Type** – Constructors do not have a return type, not even `void`.
- **Automatically Invoked** – A constructor is automatically invoked when an object of the class is created.
- **Overloading** – You can have multiple constructors in a class, as long as they have different parameter lists.

Default Constructor

A **default constructor** is one that takes no arguments. It is automatically called when an object is created without any specific initialization parameters. If no constructor is explicitly provided, the compiler generates a default constructor.

- **Default Initialization**
  - For built-in types (e.g., `int`, `double`), the compiler does not initialize these attributes, leaving them with garbage values (i.e., they are not zero-initialized).
  - For user-defined types (i.e., other class objects), the compiler calls their default constructors.
- **Attributes with Default Initializers** – If an attribute has a default initializer (e.g., `int x_{5};`), the compiler-generated default constructor will use that initializer.

Since we did not provide any constructor in the class `Robot`, the compiler provides the following default constructor `Robot()`.

Always provide your own default constructor to initialize the attributes. Do not rely on the compiler's default constructor.

```cpp
// User-defined default constructor
robotics::Robot::Robot() {
    robot_id_ = "X123";
    model_ = "UR5e";
    operational_status_ = robotics::RobotStatus::IDLE;
}
```

Which restrictions do you see with this constructor?

Parameterized Constructors

A parameterized constructor is a constructor that takes at least one parameter. A parameterized constructor, as opposed to a default constructor, allows you to initialize an object's attributes at the time of its creation by passing values as arguments.

```
className(parameter-list){
    /*constructor body*/
}
```

Example

```
robotics::Robot::Robot(const std::string& robot_id, const std::string& model) {
    robot_id_ = robot_id;
    model_ = model;
    operational_status_ = robotics::RobotStatus::IDLE;
}
```

Demonstration

```cpp
int main() {
    std::cout << "\n--- Stack Allocation Example ---\n";
    robotics::Robot stack_robot1; // default ctor
    std::cout << "Model: " << stack_robot1.get_model() << '\n';

    robotics::Robot stack_robot2("X234", "UR10e");  // parameterized ctor
    std::cout << "Model: " << stack_robot2.get_model() << '\n';

    // Unique pointer allocation example
    std::cout << "\n--- Unique Pointer Allocation Example ---\n";
    auto heap_robot1 = std::make_unique<robotics::Robot>();  // default ctor
    std::cout << "Model: " << heap_robot1->get_model() << '\n';

    auto heap_robot2 = std::make_unique<robotics::Robot>("X234", "UR10e");  // parameterized
    ↪  ctor
    std::cout << "Model: " << heap_robot2->get_model() << '\n';
}
```

ToDo

- Remove **only** the user-defined default constructor from the class `Robot`.
- Compile and run the program below.

```
int main() {
    robotics::Robot stack_robot;
}
```

Constructor Member Initializer List

When you create an object, you can use the constructor's **member initializer list** to set the initial values of the object's attributes.

```
constructor_identifier(parameters): attribute1{value1}, attribute2{value2}, ... {
    /*body of the constructor*/
}
```

This approach is often considered more efficient than assignment.

Key Points

- **Direct Initialization** – Member initializer lists initialize attributes directly, as opposed to assigning them values after they have been default-initialized. This reduces the need for a two-step process (creation and then assignment) and can be more efficient, particularly for non-trivial types.
- **Const and Reference Variables** – Attributes declared as `const` or reference must be initialized in an initializer list since they cannot be assigned to.
- **Avoiding Temporary Objects** – By using member initializers, you can avoid the creation of temporary objects that might be created during assignment. Direct initialization often avoids these temporaries.

```
1  class A {
2   public:
3    A() { std::cout << "default ctor A\n"; }
4    A(int x) : v_{x} { std::cout << "param ctor A\n"; }
5
6   private:
7    int v_;
8  };  // class A
9
10 class B {
11  public:
12   B() {
13     std::cout << "default ctor B\n";
14     a_ = A(2);  // calls param ctor for A
15   }
16
17  private:
18   A a_;  // calls default ctor for A
19 };  // class B
```

```
int main(){
    B b;
}
```

- When B is instantiated in the `main()` function, the default constructor for A (line 3) is called. This is because A is an attribute of B and needs to be initialized before B is initialized.
- The default constructor for B is then called (line 12).
  - When `a_ = A(2)` is executed (line 14), the param constructor for A is called (line 4).
- In the end, two constructors for A is called instead of one.

ToDo

Use member initialization list to initialize `a_`.

```
class DemoConstAndRef {
 public:
  DemoConstAndRef(int constant, int reference) {
    const_ = constant;  // error
    ref_ = reference;   // error
  }

 private:
  const int const_;
  int &ref_;

};  // class DemoConstAndRef
```

```
int main(){
    int some_value{43};
    DemoConstAndRef demo(10, some_value);
}
```

A constant and a reference cannot be declared and then assigned a value later, they have to be given a value during initialization.

ToDo

Use constructor member initializer list in the example above.

ToDo

Rewrite all the constructors of the `Robot` class with constructor member initializer list.

```
robotics::Robot::Robot(const std::string& robot_id, const std::string& model)
    : robot_id_{robot_id}, model_{model}, operational_status_{robotics::RobotStatus::IDLE} {
    log_activity("Robot created with ID: " + robot_id_ + ", Model: " + model_);
}
```

You can still write code in the body of the constructor.

Next Class

- Lecture 8: Object-Oriented Programming (Part 2).
  - Inheritance and modern polymorphism.
  - Virtual functions and abstract classes.
  - Smart pointer polymorphism.
  - Modern C++ 17 polymorphism patterns.