Got it! Let's approach this step by step. The goal is to bring a fresh graduate—who doesn't know **C++**—up to speed quickly so they can work effectively on an **embedded LRU unit** (with multiple cards) for both **development** and **testing**. I'll outline a **structured training plan**, mixing theory, hands-on exercises, and embedded context.

Phase 0: Assess the Junior's Background

Before starting C++ formally, gauge:

- Knowledge of basic programming concepts (variables, loops, functions).
- Understanding of electronics and microcontrollers.
- Familiarity with embedded systems concepts (registers, memory-mapped I/O, interrupts).

This helps me **customize examples** toward embedded systems rather than generic apps.

Phase 1: Fundamentals of C++ (1–2 weeks)

Objective: Learn the core syntax, logic, and program structure in C++.

Topics to cover:

1. **Basic syntax**
    - Variables, data types, constants
    - Operators and expressions
    - Input/output (`cout`, `cin`) — simple console exercises
2. **Control structures**
    - Conditional statements: `if`, `else`, `switch`
    - Loops: `for`, `while`, `do-while`
3. **Functions**
    - Definition, declaration, parameters, return types
    - Passing by value vs. reference
    - Inline functions
4. **Arrays and Strings**
    - Fixed-size arrays, `std::array`, character arrays
    - Basic string operations (`std::string`)
5. **Pointers and References**
    - Pointer basics, dereferencing, pointer arithmetic
    - References vs pointers
6. **Basic I/O for Embedded**
    - Reading/writing to variables simulating hardware registers

Hands-on mini projects:

- Simulate a **LED blinking pattern** in console using loops.
- Implement a simple **sensor reading simulation** with arrays.

Phase 2: Object-Oriented Programming in C++ (2–3 weeks)

Objective: Learn OOP concepts and apply them to embedded card architecture.

Topics to cover:

1. **Classes and Objects**
   - Class definition, member variables, member functions
   - Constructors and destructors
2. **Access specifiers**
   - `public`, `private`, `protected`
3. **Inheritance and Polymorphism**
   - Single, multilevel inheritance
   - Virtual functions
4. **Encapsulation and Abstraction**
   - Why it matters in embedded firmware for modular code
5. **Namespaces**
6. **Operator overloading** (for advanced embedded abstractions if needed)

Hands-on exercises:

- Model an **LRU card** as a class:

  ```
  class Card {
      int id;
      bool active;
  public:
      Card(int card_id);
      void activate();
      void deactivate();
      void displayStatus();
  };
  ```

- Simulate **activation/deactivation of multiple cards** in the console.
- Show **inheritance** by having different card types inherit from a base `Card` class.

Phase 3: Embedded-Specific C++ Concepts (2 weeks)

Objective: Apply C++ to real embedded programming.

Topics:

1. **Memory management**
   - Stack vs heap, `new` / `delete`
   - Importance in constrained embedded environments
2. **Volatile and const**

    - Accessing hardware registers safely
3. **Direct Register Access**
    - Using pointers to manipulate simulated hardware registers
4. **Interrupt handling**
    - Function pointers and callback functions
5. **Templates and STL (if applicable)**
    - Useful for generic functions on data structures

Hands-on exercises:

- Write **C++ classes to control virtual LEDs, buttons, and sensors**.
- Implement **a card monitoring system** with multiple LRU cards using classes and arrays.

Phase 4: C++ for Testing Firmware (2 weeks)

Objective: Enable the junior to write test code for embedded firmware using C++.

Topics:

1. **Unit testing frameworks**
    - Google Test (gtest) or Catch2
2. **Mocking hardware**
    - Use classes to simulate hardware peripherals
3. **Automated test scripts**
    - Writing test functions for card initialization, communication, error handling
4. **Logging and debugging**
    - `printf`, `std::cout`, and debug macros
5. **Test-driven development (TDD)**
    - Writing tests first, then implementing class methods

Hands-on exercises:

- Write tests for `Card` class methods:
    - Check status before/after activation
    - Simulate faulty card behavior and verify error handling
- Build a **mock LRU backplane** in code to test interaction between cards.

Phase 5: Project-Based Learning (4–6 weeks)

Objective: Integrate learning into a realistic embedded project.

Steps:

1. Assign a **small LRU system project**:
    - Multiple cards, simulated backplane, communication protocol
2. **Development tasks:**
    - Implement classes for cards and backplane

    - Simulate reading/writing registers
    - Implement error detection
3. **Testing tasks:**
    - Write unit tests and integration tests for each card
    - Validate backplane-card communication
4. **Code review & debugging:**
    - Review junior's code daily
    - Encourage incremental commits and testing

Deliverables:

- Working LRU simulation
- Test suite for each card
- Documentation of class structure and testing methodology

Phase 6: Continuous Mentoring & Best Practices

- Encourage **coding standards** (indentation, naming, commenting)
- Teach **version control** (Git) in parallel
- Introduce **embedded RTOS concepts** when comfortable
- Encourage **reading datasheets and spec documents** for real hardware implementation
- Gradually transition from **simulation to actual board testing**

Key Tips for Mentoring

- Use **analogy-based learning**: "Think of each LRU card as an object with state and behavior."
- Keep **sessions hands-on**, theory only when necessary.
- Encourage **debugging with small steps**, e.g., one card at a time.
- Gradually **increase complexity**: single card → multiple cards → full backplane → testing.

If you want, I can also **draft a full 3-month weekly C++ training schedule** specifically for LRU embedded development, with exercises, mini-projects, and testing milestones. It will map **learning → simulation → testing → actual board** step by step.

Do you want me to do that?