Plan: C++ From-Scratch Learning Path (Linear & Sequential)

A step-by-step 8-project curriculum (16-24 weeks) for Python programmers learning C++. Each project is broken into sequential phases—complete Phase 1 before Phase 2, and so on. No guessing about what to do next.

Overview: Python → C++ Key Differences

Manual memory (no GC), **static typing**, **header/implementation split**, **pointers**, **value semantics**, **templates**, and **undefined behavior**. Each project phase introduces these incrementally.

Project 0: Environment & Hello World (1-2 days)

Phase 1: Setup (30 min)

1. Install compiler (GCC/Clang/MSVC)
2. Install text editor or IDE
3. Verify installation: `g++ --version`

Phase 2: Hello World (1 hour)

1. Create `hello.cpp` with basic `main()` function
2. Add `#include <iostream>` and `std::cout << "Hello World\n";`
3. Compile manually: `g++ -std=c++17 -Wall -Wextra hello.cpp -o hello`
4. Run: `./hello`
5. **Learning checkpoint:** Understand compilation creates executable file (unlike Python's interpretation)

Phase 3: User Input (2 hours)

1. Create `greet.cpp`
2. Use `std::cin` to get user's name
3. Output personalized greeting
4. **Learning checkpoint:** Understand `>>` operator extracts from input stream

Phase 4: Multiple Functions (2-3 hours)

1. Create `multi_func.cpp`
2. Write function that adds two numbers (above `main()`)
3. Call it from `main()`
4. Move function declaration to top, definition to bottom
5. **Learning checkpoint:** Understand forward declarations and why C++ needs them

---

Project 1: Calculator (3-5 days)

Phase 1: Single Operation Calculator (Day 1)

1. Create `calculator.cpp`

2. Prompt user for two numbers (use `double`)
3. Prompt for operation (+, -, *, /)
4. Perform calculation and display result
5. **Learning checkpoint:** Integer vs float division (5/2 vs 5.0/2.0)

Phase 2: Add Input Validation (Day 1-2)

1. Add division-by-zero check with `if` statement
2. Validate operation is one of four valid choices
3. Display error messages for invalid input
4. **Learning checkpoint:** Basic error handling with conditionals

Phase 3: Add Loop & Menu (Day 2)

1. Wrap calculator in `while` loop
2. Add menu: "1. Calculate 2. Exit"
3. Use `switch` statement for menu choices
4. Let user perform multiple calculations
5. Add option to exit
6. **Learning checkpoint:** Switch statements and loop control

Phase 4: Add Exception Handling (Day 3)

1. Add `try`/`catch` blocks for division
2. `throw` exception when dividing by zero
3. Catch and display error message
4. **Learning checkpoint:** Basic exception syntax (more details later)

Phase 5: Add History Feature (Day 4-5)

1. Store last 5 calculations in array: `double results[5]`
2. Add menu option to view history
3. Update array after each calculation
4. **Learning checkpoint:** Static arrays and indexing

Phase 6: Split Into Multiple Files (Day 5)

1. Create `calculator.h` with function declarations
2. Create `calculator.cpp` with function implementations
3. Keep `main.cpp` with just `main()` function
4. Add header guards to `calculator.h`
5. Compile: `g++ main.cpp calculator.cpp -o calc`
6. **Learning checkpoint:** Header/implementation split pattern

---

Project 2: Grade Manager (5-7 days)

Phase 1: Student Class - Basic Structure (Day 1-2)

1. Create `student.h` with class declaration:

    - Private: `std::string name`, `int id`
    - Public: constructor, getters
2. Add header guard (`#pragma once`)
3. Create `student.cpp` with constructor implementation
4. Create `main.cpp` to create one `Student` object and print info
5. Create basic Makefile
6. **Learning checkpoint:** Class declaration vs implementation, compilation units

Phase 2: Add Grades Storage (Day 2-3)

1. Add `#include <vector>` to `student.h`
2. Add private member: `std::vector<double> grades`
3. Add method: `void addGrade(double grade)`
4. Add method: `double getAverage() const`
5. Test in `main.cpp`: add grades, calculate average
6. **Learning checkpoint:** `std::vector` basics, const member functions

Phase 3: Add Grade Validation (Day 3-4)

1. Modify `addGrade()` to validate grade is 0-100
2. Return `bool` (true if added, false if invalid)
3. Add method: `double getMinGrade() const`
4. Add method: `double getMaxGrade() const`
5. Test edge cases in `main.cpp`
6. **Learning checkpoint:** Input validation, returning values

Phase 4: Course Class (Day 4-5)

1. Create `course.h` and `course.cpp`
2. Course has: `std::string name`, `std::vector<Student> students`
3. Add method: `void addStudent(const Student& student)`
4. Add method: `void displayRoster() const`
5. Update Makefile to compile course files
6. Test in `main.cpp`: create course, add students, display
7. **Learning checkpoint:** Composition, pass-by-reference with `const`

Phase 5: File I/O (Day 5-6)

1. Add method to `Course`: `void saveToFile(const std::string& filename) const`
2. Use `std::ofstream` to write student data
3. Format: one line per student (name, id, grades)
4. Add method: `void loadFromFile(const std::string& filename)`
5. Use `std::ifstream` to read data back
6. Test: save, restart program, load
7. **Learning checkpoint:** File streams, parsing text data

Phase 6: Polish & Menu System (Day 6-7)

1. Add menu loop in `main.cpp`

2. Options: Add student, Add grade to student, Display roster, Save, Load, Exit
3. Implement each menu option with appropriate function calls
4. Add error handling for file operations
5. **Learning checkpoint:** Putting it all together

Project 3: CLI Task Manager (10-14 days)

Phase 1: Task Class Foundation (Day 1-2)

1. Create `task.h` and `task.cpp`
2. Add enum class: `enum class Status { TODO, IN_PROGRESS, DONE }`
3. Add enum class: `enum class Priority { LOW, MEDIUM, HIGH }`
4. Task members: `std::string title`, `std::string description`, `Status status`, `Priority priority`, `int id`
5. Constructor taking all parameters
6. Getters and setters
7. Test in `main.cpp`

Phase 2: Task Display & Comparison (Day 2-3)

1. Add method: `std::string toString() const` that formats task info
2. Overload `operator<<` as friend function for easy printing
3. Add method: `bool operator==(const Task& other) const` comparing by id
4. Test printing and comparison

Phase 3: TaskManager - Basic Storage (Day 3-5)

1. Create `task_manager.h` and `task_manager.cpp`
2. Private member: `std::vector<Task> tasks`
3. Private member: `int nextId` for auto-incrementing IDs
4. Method: `void addTask(const std::string& title, const std::string& desc, Priority p)`
5. Method: `void displayAll() const`
6. Method: `Task* findTaskById(int id)` returning pointer (nullptr if not found)
7. Set up CMake instead of Makefile: create `CMakeLists.txt`
8. Build with CMake: `mkdir build && cd build && cmake .. && make`
9. **Learning checkpoint:** Pointers for optional return values, CMake basics

Phase 4: Task Operations (Day 5-7)

1. Add method: `bool removeTask(int id)` - returns true if found and removed
2. Add method: `bool updateStatus(int id, Status newStatus)`
3. Add method: `bool updatePriority(int id, Priority newPriority)`
4. Use `std::find_if` with lambda to find tasks
5. **Learning checkpoint:** STL algorithms, basic lambdas

Phase 5: Filtering & Searching (Day 7-9)

1. Add method: `std::vector<Task> filterByStatus(Status s) const`

2. Add method: `std::vector<Task> filterByPriority(Priority p) const`
3. Add method: `std::vector<Task> search(const std::string& keyword) const`
4. Use `std::copy_if` or loops to build filtered lists
5. Test each filter function
6. **Learning checkpoint:** Working with container copies, returning containers

Phase 6: Sorting (Day 9-10)

1. Add method: `void sortByPriority()`
2. Add method: `void sortById()`
3. Use `std::sort` with custom comparators (lambdas)
4. **Learning checkpoint:** Sorting algorithms, comparators

Phase 7: File Persistence (Day 10-12)

1. Add method: `void saveToFile(const std::string& filename) const`
2. Choose simple format (CSV or custom)
3. Save all task data including enum values (convert to int)
4. Add method: `void loadFromFile(const std::string& filename)`
5. Parse file and reconstruct tasks
6. Handle file errors gracefully
7. **Learning checkpoint:** Data serialization, enum conversion

Phase 8: CLI Interface & Polish (Day 12-14)

1. Create menu system in `main.cpp`
2. Commands: add, list, complete [id], delete [id], filter, search, save, load, quit
3. Parse command-line style input (e.g., "complete 5")
4. Use `std::istringstream` for parsing commands
5. Add help command showing all available commands
6. Auto-load from default file on startup
7. Auto-save on exit
8. **Learning checkpoint:** String parsing, program flow

Project 4: Generic Data Structures (10-14 days)

Phase 1: Generic Stack (Day 1-4)

1. Create `stack.h` with template class
2. Use `std::vector<T>` internally for storage
3. Implement in header (templates require this):
    - `void push(const T& item)`
    - `void pop()` (throws exception if empty)
    - `T top() const` (throws exception if empty)
    - `bool isEmpty() const`
    - `size_t size() const`
4. Create `main.cpp` to test with `Stack<int>` and `Stack<std::string>`
5. **Learning checkpoint:** Template syntax, header-only implementation

Phase 2: Add Exception Handling (Day 4-5)

1. Create `exceptions.h` with custom exception class: `class StackEmptyException`
2. Throw in `pop()` and `top()` when empty
3. Update test code to catch exceptions
4. **Learning checkpoint:** Custom exceptions, throw/catch

Phase 3: Install Testing Framework (Day 5-6)

1. Download and set up Google Test or Catch2
2. Update `CMakeLists.txt` to include testing:

```
add_subdirectory(tests)
enable_testing()
```

3. Create `tests/test_stack.cpp`
4. Write 5-10 tests: push/pop, empty, exceptions
5. Run tests: `cd build && ctest`
6. **Learning checkpoint:** Unit testing setup, test-driven development

Phase 4: Generic Queue (Day 6-9)

1. Create `queue.h` with template class
2. Use `std::deque<T>` internally (or implement circular buffer for challenge)
3. Implement:
    - `void enqueue(const T& item)`
    - `void dequeue()` (throws if empty)
    - `T front() const` (throws if empty)
    - `T back() const` (throws if empty)
    - `bool isEmpty() const`
    - `size_t size() const`
4. Create `tests/test_queue.cpp` with comprehensive tests
5. Test with multiple types
6. **Learning checkpoint:** Queue data structure, more template practice

Phase 5: Generic Linked List (Day 9-14)

1. Create `linked_list.h`
2. Define nested struct: `template<typename T> struct Node { T data; Node* next; }`
3. Private members: `Node<T>* head`, `size_t count`
4. Implement step-by-step:
    - **Day 9-10:** Constructor, destructor (must delete all nodes!), `void insertFront(const T& item)`
    - **Day 10-11:** `void insertBack(const T& item)`, `bool remove(const T& item)`, `bool contains(const T& item) const`
    - **Day 11-12:** Copy constructor (deep copy), copy assignment operator
    - **Day 12-13:** Iterator class (nested) with `begin()` and `end()` methods

- **Day 13-14:** Write comprehensive tests

5. **Learning checkpoint:** Raw pointers, manual memory management, Rule of Three, iterators

Phase 6: Benchmarking (Day 14)

1. Create `benchmark.cpp`
2. Compare your Stack vs `std::stack` (1M pushes/pops)
3. Compare your Queue vs `std::queue`
4. Measure with `<chrono>` library
5. Display results
6. **Learning checkpoint:** Performance measurement, realistic expectations

---

Project 5: Memory Pool Allocator (14-21 days)

Phase 1: Understand the Heap (Day 1-3)

1. Create `pointer_basics.cpp` for experiments
2. Experiment with `new` and `delete`:

```
int* p = new int(42);
delete p;
```

3. Try `new[]` and `delete[]` with arrays
4. Intentionally create memory leak (allocate without deleting)
5. Run with Valgrind: `valgrind --leak-check=full ./program`
6. Observe leak report
7. Fix leak and verify with Valgrind
8. **Learning checkpoint:** Heap allocation basics, memory leak detection

Phase 2: Pointer Arithmetic & Arrays (Day 3-5)

1. Create `pointer_arithmetic.cpp`
2. Allocate array: `int* arr = new int[10]`
3. Access elements with `arr[i]` and `*(arr + i)`
4. Print addresses to see memory layout
5. Understand pointer arithmetic (moves by sizeof(T))
6. Delete array properly: `delete[] arr`
7. **Learning checkpoint:** Pointer arithmetic, array/pointer relationship

Phase 3: Rule of Three - Manual Resource Management (Day 5-8)

1. Create `resource_holder.h` and `resource_holder.cpp`
2. Class holds dynamically allocated int array
3. Implement:
    - **Day 5-6:** Constructor allocating array with `new[]`
    - **Day 6:** Destructor deleting array with `delete[]`

- **Day 6-7:** Copy constructor doing deep copy
- **Day 7-8:** Copy assignment operator (check self-assignment!)

4. Write tests creating copies, checking independence
5. Run with Valgrind to ensure no leaks
6. **Learning checkpoint:** Rule of Three, deep vs shallow copy

Phase 4: Fixed-Size Memory Pool - Core (Day 8-12)

1. Create `memory_pool.h` and `memory_pool.cpp`
2. Constructor: allocate one large block with `new char[poolSize]`
3. Maintain free list using intrusive linked list:

```
struct FreeNode { FreeNode* next; };
```

4. Implement `void* allocate(size_t size)`:
   - **Day 8-9:** Pop from free list
   - Return nullptr if no space
5. Implement `void deallocate(void* ptr)`:
   - **Day 9-10:** Push back to free list
6. Destructor: delete the large block
7. **Learning checkpoint:** Custom allocators, free list data structure

Phase 5: Object Pool (Day 12-15)

1. Create `object_pool.h` - template class
2. Specialize for single object type: `template<typename T> class ObjectPool`
3. Use placement new: `new (ptr) T(args...)`
4. Implement `T* allocate()` using memory pool
5. Implement `void deallocate(T* ptr)`:
   - Call destructor explicitly: `ptr->~T()`
   - Return memory to pool
6. Test with custom class (e.g., `Student` from Project 2)
7. **Learning checkpoint:** Placement new, explicit destructor calls

Phase 6: Benchmarking & Stress Testing (Day 15-18)

1. Create `benchmark_allocator.cpp`
2. Test 1: Allocate/deallocate 1M objects sequentially
   - Compare pool vs raw `new`/`delete`
   - Measure time with `<chrono>`
3. Test 2: Random allocate/deallocate pattern
4. Test 3: Allocate many, deallocate all (fragmentation test)
5. Run with Valgrind/AddressSanitizer
6. **Learning checkpoint:** Performance characteristics, profiling

Phase 7: Move Semantics (Day 18-21)

1. Add to `resource_holder.cpp`:
    - **Day 18-19:** Move constructor (steals resources)
    - **Day 19-20:** Move assignment operator
    - Use `std::move` in tests
2. Update to Rule of Five
3. Benchmark copy vs move operations
4. **Learning checkpoint:** Rvalue references, move semantics, Rule of Five

---

Project 6: Multi-threaded File Downloader (14-21 days)

Phase 1: Smart Pointers Basics (Day 1-3)

1. Create `smart_ptr_demo.cpp` for experiments
2. **Day 1:** Use `std::unique_ptr<int>`:
    - `auto p = std::make_unique<int>(42)`
    - Observe automatic deletion (no explicit delete needed)
    - Try transferring ownership with `std::move`
3. **Day 2:** Use `std::shared_ptr<int>`:
    - Create shared pointer
    - Make copies, observe reference counting
    - Check `use_count()`
4. **Day 3:** Use `std::weak_ptr`:
    - Break circular reference problem
5. **Learning checkpoint:** Smart pointer types, RAII for memory

Phase 2: Threading Basics (Day 3-6)

1. Create `thread_basics.cpp`
2. **Day 3-4:** Create thread executing simple function:

```
void worker() { /* ... */ }
std::thread t(worker);
t.join();
```

3. **Day 4:** Pass parameters to thread function
4. **Day 5:** Create multiple threads in loop, join all
5. **Day 5-6:** Use `std::this_thread::sleep_for` to simulate work
6. **Learning checkpoint:** Thread creation, joining, basic concurrency

Phase 3: Mutex & Race Conditions (Day 6-9)

1. Create `race_condition.cpp`
2. **Day 6-7:** Create race condition:
    - Shared counter
    - Multiple threads incrementing
    - Observe incorrect final value

3. **Day 7-8:** Fix with `std::mutex` and `std::lock_guard`
4. **Day 8-9:** Try `std::unique_lock` for more control
5. Run with ThreadSanitizer: `g++ -fsanitize=thread`
6. **Learning checkpoint:** Race conditions, mutual exclusion

Phase 4: Lambda Expressions (Day 9-10)

1. Create `lambda_demo.cpp`
2. **Day 9:** Basic lambda: `auto f = []() { /* ... */ };`
3. Capture by value: `[x]`
4. Capture by reference: `[&x]`
5. Capture all: `[=]` or `[&]`
6. **Day 10:** Use lambda with `std::thread`
7. **Learning checkpoint:** Lambda syntax, captures

Phase 5: Simple HTTP Downloader (Day 10-13)

1. Install libcurl
2. Create `downloader.h` and `downloader.cpp`
3. **Day 10-11:** Write `bool downloadFile(const std::string& url, const std::string& outputPath)`
4. Use libcurl to fetch URL and save to file
5. **Day 11-12:** Add error handling
6. **Day 12-13:** Test with real URLs
7. Update `CMakeLists.txt` to link libcurl
8. **Learning checkpoint:** External library integration, HTTP basics

Phase 6: Thread Pool (Day 13-16)

1. Create `thread_pool.h` and `thread_pool.cpp`
2. **Day 13-14:** Class members:
    - `std::vector<std::thread> workers`
    - `std::queue<std::function<void()>> tasks` (task queue)
    - `std::mutex queueMutex`
    - `std::condition_variable condition`
    - `bool stop`
3. **Day 14-15:** Constructor: create N worker threads that:
    - Wait on condition variable
    - Pop task from queue
    - Execute task
4. **Day 15-16:** Method: `void enqueue(std::function<void()> task)`
5. Destructor: signal stop, join all threads
6. **Learning checkpoint:** Condition variables, producer-consumer pattern

Phase 7: Download Manager (Day 16-19)

1. Create `download_manager.h` and `download_manager.cpp`
2. **Day 16-17:** Class has:

    - `ThreadPool pool`
    - `std::map<int, DownloadInfo>` tracking downloads (use mutex!)
    - `int nextId`
3. **Day 17:** Method: `int addDownload(const std::string& url, const std::string& output)`
    - Returns download ID
    - Enqueues task to thread pool
4. **Day 17-18:** Method: `DownloadStatus getStatus(int id)`
5. **Day 18-19:** Track progress (use `std::atomic<int>` for progress percentage)
6. **Learning checkpoint:** Concurrent data structures, atomic operations

Phase 8: CLI & Polish (Day 19-21)

1. Create `main.cpp` with menu
2. **Day 19-20:** Commands: add [url], status [id], list, wait, quit
3. **Day 20:** Display progress bars for active downloads
4. **Day 20-21:** Add pause/resume functionality (challenge: need to modify download function)
5. Stress test: download 20 files simultaneously
6. **Learning checkpoint:** Putting concurrency together

Project 7: 2D Game Engine (21-30 days)

Phase 1: SDL2 Setup & Window (Day 1-2)

1. Install SDL2
2. **Day 1:** Create basic `main.cpp` that:
    - Initializes SDL
    - Creates window
    - Event loop (handle quit event)
    - Cleans up SDL
3. **Day 2:** Add renderer, clear screen with color, present
4. **Learning checkpoint:** SDL basics, game loop structure

Phase 2: Entity Base Class (Day 2-4)

1. Create `entity.h` and `entity.cpp`
2. **Day 2-3:** Abstract base class:

```
class Entity {
public:
    virtual ~Entity() = default;
    virtual void update(float deltaTime) = 0;
    virtual void render(SDL_Renderer* renderer) = 0;
};
```

3. **Day 3-4:** Add members: position (x, y), velocity, size
4. Add getters/setters
5. **Learning checkpoint:** Abstract base classes, virtual functions

Phase 3: Concrete Entity - Player (Day 4-6)

1. Create `player.h` and `player.cpp` inheriting from `Entity`
2. **Day 4-5:** Implement `update()`:
   - Apply velocity to position
   - Keep in bounds
3. **Day 5-6:** Implement `render()`:
   - Draw rectangle with SDL_RenderFillRect
4. Test in main: create player, update, render in game loop
5. **Learning checkpoint:** Inheritance, virtual function override

Phase 4: Input Handling (Day 6-8)

1. Create `input_manager.h` and `input_manager.cpp`
2. **Day 6-7:** Singleton pattern:

```
class InputManager {
private:
    InputManager() = default;
public:
    static InputManager& getInstance();
    // ...
};
```

3. **Day 7:** Track key states: `std::map<SDL_Keycode, bool> keyStates`
4. **Day 7-8:** Methods: `void update(SDL_Event& event)`, `bool isKeyPressed(SDL_Keycode key)`
5. Connect to player: move on WASD/arrow keys
6. **Learning checkpoint:** Singleton pattern, input handling

Phase 5: Entity Manager & Polymorphism (Day 8-11)

1. Create `entity_manager.h` and `entity_manager.cpp`
2. **Day 8-9:** Store entities: `std::vector<std::unique_ptr<Entity>> entities`
3. **Day 9-10:** Methods:
   - `void addEntity(std::unique_ptr<Entity> entity)`
   - `void updateAll(float deltaTime)` - calls update on each
   - `void renderAll(SDL_Renderer* renderer)` - calls render on each
4. **Day 10-11:** Update main to use EntityManager
5. **Learning checkpoint:** Polymorphism in action, managing object lifetimes

Phase 6: Enemy Class (Day 11-13)

1. Create `enemy.h` and `enemy.cpp` inheriting from `Entity`
2. **Day 11-12:** Simple AI: move toward player
3. **Day 12-13:** Implement in `update()`:
   - Get player position
   - Calculate direction vector
   - Move toward player

4. Add enemies to EntityManager
5. **Learning checkpoint:** More inheritance practice, basic AI

Phase 7: Collision Detection (Day 13-16)

1. Create `physics.h` and `physics.cpp`
2. **Day 13-14:** Function: `bool checkCollision(const Entity& a, const Entity& b)`
   - AABB (axis-aligned bounding box) collision
3. **Day 14-15:** Add collision callbacks to Entity:

```
virtual void onCollision(Entity* other) {}
```

4. **Day 15-16:** In game loop:
   - Check all entity pairs for collision
   - Call `onCollision()` on both
5. Make player and enemies respond to collisions
6. **Learning checkpoint:** Collision detection, callbacks

Phase 8: Resource Manager (Day 16-19)

1. Install SDL_image for texture loading
2. Create `resource_manager.h` and `resource_manager.cpp`
3. **Day 16-17:** Singleton managing resources:
   - `std::map<std::string, SDL_Texture*> textures`
4. **Day 17-18:** Methods:
   - `SDL_Texture* loadTexture(const std::string& path, SDL_Renderer* renderer)`
   - Cache textures (load once, return same pointer on subsequent calls)
   - `void cleanup()` - free all textures
5. **Day 18-19:** Update Entity to use textures instead of rectangles
6. **Learning checkpoint:** Resource management, texture rendering

Phase 9: Game States & Menu (Day 19-22)

1. Create `game_state.h` with enum: `enum class GameState { MENU, PLAYING, PAUSED, GAME_OVER }`
2. **Day 19-20:** Create `menu_state.h` and `menu_state.cpp`
3. **Day 20-21:** Implement menu:
   - Display "Start Game", "Quit"
   - Handle click or key selection
4. **Day 21-22:** Add state management to main game loop
5. Switch between menu and playing states
6. **Learning checkpoint:** State pattern, game flow

Phase 10: Particle System (Day 22-25)

1. Create `particle.h` and `particle.cpp` (inherits Entity)
2. **Day 22-23:** Particle has:

    - Lifetime counter
    - Fading alpha
3. **Day 23-24:** Create `particle_system.h`:
    - Pool of particles (object pool pattern)
    - Method: `void emit(float x, float y, int count)`
4. **Day 24-25:** Emit particles on enemy death
5. **Learning checkpoint:** Object pool pattern in practice, visual effects

Phase 11: Audio System (Day 25-27)

1. Install SDL_mixer
2. Create `audio_manager.h` and `audio_manager.cpp`
3. **Day 25-26:** Singleton managing sounds:
    - `std::map<std::string, Mix_Chunk*> sounds`
    - Method: `void playSound(const std::string& name)`
4. **Day 26-27:** Add sounds:
    - Player shoot
    - Enemy death
    - Background music
5. **Learning checkpoint:** Audio integration, event-based sound

Phase 12: Polish & Complete Game (Day 27-30)

1. **Day 27:** Add scoring system
2. **Day 28:** Add multiple levels (increase difficulty)
3. **Day 28:** Add health system to player
4. **Day 29:** Add game over screen
5. **Day 29:** Save/load high scores
6. **Day 30:** Bug fixing, optimization, code cleanup
7. Write comprehensive README with build instructions
8. **Learning checkpoint:** Project completion, documentation

---

Build System Progression

Projects 0-1: Direct `g++` commands
Project 2: Basic Makefile
Projects 3-7: CMake (update `CMakeLists.txt` each phase as needed)

Testing Progression

Projects 0-3: Manual testing after each phase
Project 4: Automated testing introduced (Google Test/Catch2)
Projects 5-7: Write tests alongside implementation

Time Estimates

Fast Track (15-20 hrs/week): 16 weeks
Thorough Track (10-15 hrs/week): 24 weeks

Further Considerations

1. **IDE preference?** VS Code (lightweight), CLion (powerful), or Visual Studio (Windows)?
2. **C++ standard?** C++17 (stable) or C++20 (modern features)?
3. **Testing framework?** Google Test (industry standard) or Catch2 (simpler)?
4. **Compiler?** GCC, Clang, or MSVC?