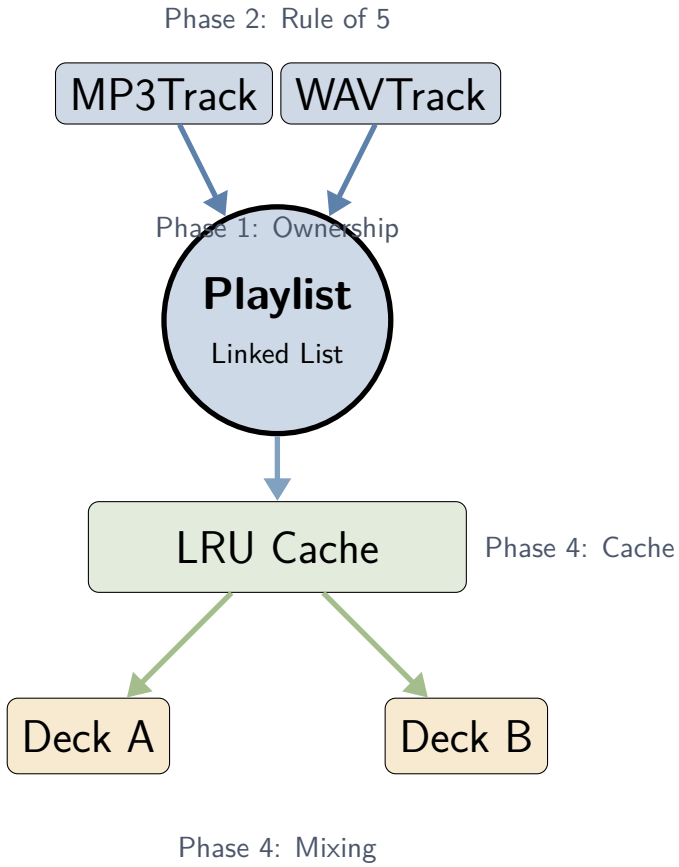
Systems Programming 202.1.2031

Winter 2026 Assignment 1

Lotem Sakira, Ahmed Massalha, Dayan Badalbaev

DJ Track Session Manager: C++ Memory Management



Ben-Gurion University
of the Negev

Contents

1 Introduction

Welcome to the world of advanced C++ memory management!

This assignment explores **memory ownership**, a core concept that separates professional C++ developers from beginners, through four progressive phases. You'll work with a small but realistic application that demonstrates how memory management challenges arise in real-world software development.

In this assignment, you will face one of the most important design questions in memory management: Who owns the memory? Which entity is responsible for managing it, and why? In this codebase, one approach can lead to code that is messy, hard to debug, and prone to errors, while another can make your program far easier to analyze and maintain.

Throughout the assignment, you will build a comprehensive DJ Track Session Manager system, inspired by professional tools such as rekordbox or Serato DJ.

The assignment is divided into four progressive phases, each building on the last:

- **Phase 1**: A Broken Playlist: Fix intentionally broken code
- **Phase 2**: Master of 5 and Polymorphism: Implement Rule of 5 and Polymorphism
- **Phase 3**: Pointer Wrapper: Create a class that wraps pointers
- **Phase 4**: System Integration: Bring everything together

By the end of this assignment, you will understand how professional software manages resources, why memory leaks are catastrophic in production systems, and how modern C++ provides elegant solutions to these challenges.

Don't Panic. While this document might seem long, it is carefully broken into phases to make your work manageable.

Memory management in C++ can feel overwhelming at first, but it is one of the most rewarding skills you can develop. This assignment is designed to challenge you while guiding you through each concept step by step.

Take your time, read the hints carefully, and test frequently. If your code doesn't compile or crashes, that's normal, as debugging is part of the learning process.

Notes on using LLMs and agents You may be tempted to use LLMs or coding agents to solve the assignment. Remember, however, that this exercise is designed to deepen your understanding of the memory management chapter in the course. If you simply let a tool solve the tasks for you, you will miss the point of the assignment **and it will be obvious to us that you did**. If you choose to use such tools, do so wisely: use them to understand the material, to clarify concepts, or to explore different design choices, but not to skip the work itself.

Good luck and happy coding!

2 Submission Instructions

- Make sure your code compiles without errors, warnings, and **without memory leaks**, using the provided Makefile.
- **Error messages must be printed using the `[ERROR] prefix`.**
- We recommend commenting on your code and explaining your design choices. This will be helpful during the discussion with the graders.
- **Submit a single archive containing your complete project in a `.zip` file named using the format `ID1_ID2.zip`, where `ID1` and `ID2` are your IDs. All phases must be included in the same codebase, and functionality must not be broken.**
- We strongly recommend using Git and committing your changes regularly. This will help you track progress and maintain working versions when things go wrong.
- The theoretical questions in this assignment are **not for submission**, but you are expected to know the answers for grading discussions.
- Submission is allowed **only in pairs** and **only via Moodle**. Email submissions will not be accepted.
- Before submitting, make sure the system's output matches the expected output, build your project, and check your code in the labs before submitting:

  ```
  make clean; make release
  ```

  After that, clean your project by running the following command in your project directory:

  ```
  make clean
  ```

  This will remove all compiled files.
- Test your submission with valgrind, a memory debugging tool, to ensure there are no memory leaks:

  ```
  make test-leaks  # If valgrind is available
  ```
- **Important**: The provided tests are basic verification only. You must thoroughly test your code throughout all phases and consistently check for memory leaks after implementing each phase.
- The archive should include the following structure (and nothing more):

  ```
  .
  +-- bin (directory)
  +-- include (directory)
  |   +-- *.h
  +-- README.md (optional)
  +-- src (directory)
  |   +-- *.cpp
  \-- Makefile
  ```
- **Keep implementation in *.cpp files and declarations in *.h files** unless otherwise specified.
- Help with the assignment will be provided via the course forum in Moodle for general questions and discussions or during office hours for specific questions about your code. Please email us in advance and check the forum before coming.

3 Phase 0: Setup and Environment

Start by downloading the DJ Track Library Manager skeleton code and setting up your development environment. This ensures you have all the necessary tools needed for memory debugging and leak detection.

Setup your development environment

1. Download the skeleton code from the course repository:

   ```
   git clone https://github.com/bguspl/SPL25-Assignment1 dj-assignment
   cd dj-assignment
   ```

   Note: You can also fork the repository or clone it through your IDE's GUI.
2. Check the project structure: You should see `.devcontainer/`, `src/`, `include/`, `Makefile`, and `README.md`.
3. Open the project in Visual Studio Code and reopen the project in a container using the provided `.devcontainer/` configuration. This ensures a consistent development environment with all necessary tools pre-installed.
4. Build and run the skeleton to see the current state:

   ```
   make test-leaks
   ```
5. You should receive a successful compilation, along with memory leak reports and "TODO" messages. This is expected as your task is to fix these issues.
6. If you are working on a native Linux machine (i.e, in dual boot) or inside a virtual machine, make sure the required tools are installed:

   ```
   # Ubuntu/Debian
   sudo apt-get install build-essential g++ valgrind gdb

   # Or use the provided target
   make install-deps
   ```

For submission: Nothing to submit in this phase.

Note on DEBUG mode

The Makefile includes a `DEBUG` target that compiles the code with debug symbols and without optimizations. This is useful for debugging and detecting memory leaks. As you progress, the output might become verbose. To reduce noise, consider using the `DEBUG` target: `make debug`. **However, ensure that your final submission compiles and runs correctly in both debug and release modes. leave exisiting debug tags as is.** For example, you can wrap debug information around debug prints in your code using:

```
#ifdef DEBUG
    // Debug-specific code here
#endif
```

4 Phase 1: A Broken Playlist

In this phase, your task is to fix memory leaks in the **Playlist** class while learning ownership semantics and memory debugging tools. The implementation is nearly complete, but has intentional memory leaks.[1]

Constraints: No standard library usage. Cannot change existing public interface.

Key Concepts: The `Playlist` manages a linked list of `PlaylistNode` objects, each referencing an `AudioTrack`. You must decide who owns what: should nodes own tracks, or just reference them? Your destructor, `add_track()`, and `remove_track()` must implement this consistently.

About ownership and memory management In C++, ownership refers to the decision of which class is responsible for deleting dynamically allocated resources. When a class owns a resource, its destructor must release it.

Owning and Borrowing In C++, "owning" a resource means that a class is responsible for its lifetime, including deallocation. "Borrowing" means that a class can use a resource without taking ownership. This distinction is crucial for effective memory management.

Phase 1 — Fix the Playlist class

1. Run the test program and observe the memory leak warnings:

   ```
   make test-leaks
   ```

   You should see warnings related to memory leaks and incomplete destructors. Examine those errors,...
2. Examine `src/Playlist.cpp` and `include/Playlist.h`:
   - Can you figure out what is missing or broken in the class implementation?
   - **Hint:** Sketch the memory diagram of the `Playlist` class and its linked list nodes based on the test code.
3. Fix the errors:
   - By now, you probably identified the memory leaks and their causes. You should notice the resources involved in this phase.
   - Your task is to decide on an ownership model for the resources and implement it consistently. A critical design choice you should tackle is whether `PlaylistNode` should own the `AudioTrack` objects (and be responsible for deleting them), or should it only reference them (letting someone else manage their lifetime)?
   - **Hint**: The `del` variable in `main.cpp` suggests there are different valid ownership approaches. You may need to modify it according to your preferences.
   - **Note:** You might still get 2 memory leaks from AudioTrack objects themselves - these will be fixed after properly implementing Phase 2.
4. Verify your fixes with memory tools:

[1] Real-world software often has similar issues that cause production crashes and resource exhaustion.

```
make test-leaks      # Valgrind verification
```

Common Pitfall: Inconsistent Ownership

Both ownership approaches can work, but you must be consistent:

- If `PlaylistNode` owns tracks: its destructor deletes them
- If `PlaylistNode` borrows tracks: `main()` must delete them

Your choice affects `remove_track()`, copy operations, and later phases!

For submission: Provide your modified `Playlist.cpp` and `Playlist.h` with fully functional memory management.

Questions — not for submission

1. What is the difference between ownership and borrowing in C++ code? How does this apply to the `Playlist` class?
2. What is the Rule of 3? What is the Rule of 5? When should we use the rule of 3, and when should we use the rule of 5?
3. What problems occur when copying a `Playlist` object without proper Rule of 3 implementations?
4. In the linked list context, why is it important to manage memory carefully when removing nodes?
5. Why might a class prefer to delete its copy/move constructors and operators instead of implementing deep copying?

5 Phase 2: Master of 5 and Polymorphism

In this phase, you will implement the complete **Rule of 5** for the `AudioTrack` class hierarchy and implement its derived classes. This is the heart of modern C++ memory management and is essential for building robust applications that handle dynamic resources safely and efficiently. The Rule of 5 ensures that objects managing dynamic memory can be safely copied, moved, and destroyed without leaks or undefined behavior. Professional software, such as audio applications, video games, and system software, relies heavily on these constructs to prevent crashes and ensure stability.

About `AudioTrack` and Its Dynamic Resource The `AudioTrack` class serves as a base for different audio formats. Its key feature is a dynamically allocated array, `waveform_data`, which simulates storing raw audio samples.

What is Waveform Data? In real audio software, waveform data represents the audio signal as amplitude values sampled at regular intervals (e.g., 44,100 times per second for CD quality). Each sample is a value between -1.0 and 1.0, representing the amplitude of the sound wave. Professional DJ software analyzes this data for beat detection, key analysis, and visual waveform display.

Because this resource is managed via a raw pointer, you must explicitly define how it is handled during the object's lifecycle (construction, copying, moving, and destruction). Failure to do so is a primary source of bugs in C++ programs.

Your task is to implement the five special member functions that give you complete control over this resource.

Phase 2 — Implement the Complete Rule of 5 and Polymorphism

1. **Analyze the Code**: Examine the code in `include/AudioTrack.h`, `include/MP3Track.h`, `include/WAVTrack.h` and their respective `*.cpp` files.
2. **Analyze the Resource**: Examine the `AudioTrack` class in `src/AudioTrack.cpp`.
   - Identify where the `waveform_data` array is allocated and locate the five empty function skeletons that constitute the Rule of 5.
   - Based on the lecture notes, explain to yourself why the default compiler-generated versions of these functions would be incorrect for this class.
3. **Implement the Destructor**: Complete the `~AudioTrack()` function. It must deallocate the memory used by `waveform_data` to prevent memory leaks.
4. **Implement Deep Copy Semantics**:
   - **Copy Constructor**: Implement `AudioTrack(const AudioTrack& other)`. It must perform a **deep copy** by allocating new memory for `waveform_data` and copying the contents from the source object.
   - **Copy Assignment Operator**: Implement `operator=(const AudioTrack& other)`. This function must handle self-assignment (e.g., `track = track;`), clean up its own existing resources, and then perform a deep copy of the other object's

data.

5. **Implement Move Semantics** for efficiency:
   - **Move Constructor**: Implement `AudioTrack(AudioTrack&& other) noexcept`. Instead of copying, it should "steal" the pointer and resources from the source object, leaving the source in a valid (but empty) state.
   - **Move Assignment Operator**: Implement `operator=(AudioTrack&& other) noexcept`. It should handle self-assignment, deallocate its own resources, and then steal the resources from the source object.
6. **Implement Virtual Functions in Derived Classes**: In `MP3Track.cpp` and `WAVTrack.cpp`, you must provide concrete implementations for the pure virtual functions inherited from `AudioTrack`. This demonstrates polymorphism.
   - `load()`: Simulate track loading with format-specific operations:
     - **MP3Track requirements**:
       (a) Print loading message: `[MP3Track::load] Loading MP3: "<title>" at <bitrate> kbps...`
       (b) If `has_id3_tags` is `true`, print: `→ Processing ID3 metadata (artist info, album art, etc.)...`, Otherwise print: `→ No ID3 tags found.`
       (c) Print: `→ Decoding MP3 frames...`
       (d) Print: `→ Load complete.`
     - **WAVTrack requirements**:
       (a) Print loading message: `[WAVTrack::load] Loading WAV: "<title>" at <sample_rate>Hz/<bit_depth>bit (uncompressed)...`
       (b) Calculate estimated file size in bytes:
       `size = duration_seconds * sample_rate * (bit_depth / 8) * 2`
       (Use `long long` type for the calculation)
       (c) Print: `"  → Estimated file size: <size> bytes"`
       (d) Print: `"  → Fast loading due to uncompressed format."`
   - `analyze_beatgrid()`: Simulate format-specific beat detection analysis:
     - **MP3Track requirements**:
       (a) Print analysis header: `[MP3Track::analyze_beatgrid] Analyzing beat grid for: "<title>"`
       (b) Calculate estimated beats: `beats = (duration_seconds / 60.0) * bpm`
       (c) Calculate compression precision factor based on bitrate:
       * Formula: `precision_factor = bitrate / 320.0`, this simulates how compression artifacts affect beat detection accuracy
       (d) Print: `"  → Estimated beats: <beats_estimated>  → Compression precision factor: <precision_factor>"`

- **WAVTrack requirements**:
  (a) Print analysis header: `[WAVTrack::analyze_beatgrid] Analyzing beat grid for: "<title>"`
  (b) Calculate estimated beats: `beats = (duration_seconds / 60.0) * bpm`
  (c) Precision factor is always 1 (uncompressed audio has perfect precision)
  (d) Print: `"  → Estimated beats: <beats_estimated>  → Precision factor: 1 (uncompressed audio)"`

- `get_quality_score() const`: Calculate and return a quality score as a `double`.
  - **MP3Track requirements**:
    (a) Calculate base score: `(bitrate / 320.0) * 100.0`
    * 320 kbps = 100 points (maximum quality)
    * Lower bitrates scale proportionally
    (b) Apply bonus: Add 5 points if `has_id3_tags` is `true`
    (c) Apply penalty: Subtract 10 points if `bitrate < 128`
    (d) Clamp final score to range [0.0, 100.0] and return it.
  - **WAVTrack requirements**:
    (a) Start with a base score of 70 points
    (b) Add 10 points if `sample_rate >= 44100` (CD quality)
    (c) Add 5 more points if `sample_rate >= 96000` (high-res audio)
    (d) Add 10 points if `bit_depth >= 16` (CD quality)
    (e) Add 5 more points if `bit_depth >= 24` (professional quality)
    (f) Cap maximum score at 100 points and return the final score

  Note: This method does **not** print output. It only calculates and returns the score value. The score is used internally by the system for quality assessment.

- `clone() const`: This is crucial for polymorphic copying. It should return a new, dynamically allocated copy of the current object.
  - The skeleton uses `PointerWrapper<AudioTrack>` as return type. Think about why this is a good choice for memory safety and ownership semantics.

Memory Testing Requirement

After implementing the Rule of 5, run comprehensive leak detection:

```
make clean && make test-leaks
```

You should see **zero memory leaks** both from playlist nodes (Phase 1) and AudioTrack waveform data (Phase 2).

For submission: `AudioTrack.cpp`, `MP3Track.cpp`, and `WAVTrack.cpp` with your complete implementations.

Questions — not for submission

1. Why is the Rule of 5 necessary for the `AudioTrack` class but might not be for a class that only contains primitive types (like `int` or `double`)?
2. What specific problems (e.g., double free, memory leak, dangling pointer) would occur if you only implemented the destructor but not the copy constructor and copy assignment operator?
3. Explain the difference between deep copying and shallow copying in the context of the `waveform_data` member.
4. Why are move semantics considered an optimization? What is the performance difference between copying and moving an `AudioTrack` object?
5. What does it mean to "leave the source object in a valid, destructible state" after a move? Why is setting the source's pointer to `nullptr` a common practice?
6. How does `std::move` relate to rvalue references, and why does it enable move semantics without actually "moving" anything itself?
7. In the copy/move assignment operator, why is checking for self-assignment (e.g., `if (this != &other)`) a critical first step?
8. How does the virtual destructor in the `AudioTrack` base class ensure that derived class objects (`MP3Track`, `WAVTrack`) are destroyed correctly when managed via a base class pointer?
9. Explain the purpose of the `clone()` method. Can you just use the copy constructor to create a polymorphic copy?
10. What is the "copy-and-swap" idiom, and how could it be used to implement the copy assignment operator for `AudioTrack`? What are its advantages?

6 Phase 3: Pointer Wrapper

In this phase, you will implement a template class called `PointerWrapper`. This class will wrap a raw pointer and provide additional functionality through operator overloading and careful resource handling.

This exercise is designed to deepen your understanding of pointer management, ownership semantics, and operator overloading in C++. By creating a custom pointer wrapper, you will gain insight into how we can manage dynamic memory safely and effectively in a single context, applicable to many scenarios.

Your task is to examine the existing code structure and determine what functionality this wrapper needs to provide. Through this exercise, you will discover important principles about pointer management and ownership in C++.

Guiding Questions for Your Implementation Before you begin coding, consider these fundamental design questions: What happens to dynamically allocated memory if nobody deletes it, and how can a wrapper class prevent memory leaks through RAII? What does it mean for a wrapper to "own" a pointer—can two wrappers safely share ownership, and how should ownership transfer between objects? Why is copying a pointer wrapper dangerous compared to moving ownership, and when would you choose one over the other? Consider how your wrapper should behave like a regular pointer through operator overloading, what operations users typically need (dereferencing, member access, raw pointer access), and how to safely give up ownership without destroying the managed object.

Looking ahead to Phase 4, think about how this wrapper will facilitate ownership transfer between system layers: Why would `release()` be particularly useful at cache-to-mixer boundaries? When would `reset()` be needed to replace a managed object? How does your design ensure that each resource has exactly one owner at any given time, preventing double deletion while enabling safe transfers across service boundaries? **Hint:** Check the additional material to understand some methods better.

Phase 3 — Implement PointerWrapper Template Class

1. **Examine the skeleton code**:
   - Study `include/PointerWrapper.h` and understand the template structure
   - Note which operations are explicitly deleted and think about why
   - Find all TODO functions that need implementation
   - Run the Phase 3 test to see what behavior is expected:

     ```
     make test
     ```
2. **Analyze the requirements**:
   - What should happen when a `PointerWrapper` object goes out of scope?
   - How should the wrapper behave when moved from one object to another?
   - What does it mean that copy operations are deleted?
   - Why might someone want to wrap a raw pointer in a class?

3. **Implement the core functionality**:
   - Complete the destructor - decide what should happen to the wrapped pointer
   - Implement move constructor - how should ownership transfer work?
   - Implement move assignment operator - handle self-assignment and cleanup
   - Consider the principles you learned from the Rule of 5 in Phase 2
4. **Implement access operations**:

```
T& operator*() const;
T* operator->() const;
T* get() const;
```

5. **Implement utility functions**:

```
T* release();
void reset(T* new_ptr);
explicit operator bool() const;
```

6. **Complete the global functions**:
   - Implement the global `swap` function for `PointerWrapper`
   - **Note:** While `make_pointer_wrapper` is provided, and `swap` is optional, understanding their purpose helps you grasp modern C++ idioms
   - Think about why these utilities might be useful for move operations and exception safety
7. **Test and analyze your implementation**:

```
make test-leaks
```

   - Does your implementation prevent memory leaks?
   - What happens when multiple wrappers try to manage the same pointer?
   - How does your wrapper ensure only one object owns each pointer?

For submission: Complete `PointerWrapper.h` with all TODO functions implemented. Your implementation should demonstrate understanding of ownership principles and RAII.

Questions — not for submission

1. After implementing your wrapper, what advantages does it provide over using raw pointers directly?
2. What does "ownership" mean in the context of your pointer wrapper? How does it relate to RAII?
3. Why are copy operations deleted, and what problems would arise if they weren't?
4. What is the difference between `release()` and `reset()` in your implementation? When would you use each?
5. How does your wrapper help prevent common pointer-related bugs (double-free, memory leaks, dangling pointers)?
6. What happens if you dereference your wrapper when it contains a null pointer? Should

you check before dereferencing?

7. Why is the boolean conversion operator marked `explicit`? What problems does this prevent?
8. How does your implementation relate to the RAII principle you learned about in previous phases?
9. How would you use this wrapper to transfer a track from the cache layer to the mixer layer in Phase 4?
10. Why might `release()` be preferable to `get()` when crossing service boundaries in Phase 4?

7 Phase 4: DJ Controller System

In this final phase, you integrate all previous work into a cohesive, service–oriented DJ Controller simulation that mirrors real hardware constraints (limited cache, deck switching, BPM tolerance). You will refine ownership boundaries and apply polymorphic cloning without introducing memory leaks.

Earlier phases gave you implementations (Playlist, AudioTrack hierarchy, PointerWrapper). In Phase 4, you focus on composing these pieces: building a canonical library, caching cloned instances, and performing deck operations with instant transitions. The difficulty is mostly conceptual (data flow and ownership) rather than algorithmic. Work incrementally: library building $\rightarrow$ playlist materialization $\rightarrow$ cache access $\rightarrow$ deck loading.

7.1 System Architecture Overview

The system follows a service-oriented architecture with a clear separation of concerns. It is divided into four main layers, each with distinct responsibilities.

Your system takes a configuration file (e.g., config.txt) that sets parameters, defines the track library, and lists playlists.

The output is a comprehensive console report that summarizes the DJ session, including loaded tracks, analysis results, and any errors (see the running example below).

The central orchestrator is the DJSession class, which manages three distinct services: the `DJLibraryService` handles playlists, the `DJControllerService` oversees caching, and the `MixingEngineService` runs deck operations.

Note: You will not work with actual audio data, but will simulate the management and analysis of audio tracks.

7.1.1 Configuration File Format

Your implementation will work with the following file format:

Configuration File (`dj_config.txt`):

```
# DJ Track Library Manager
# Application Settings
app_name={String}
version={major}.{minor}
# Track Library Definition
# Format: library_track_i:{MP3,WAV},title,{artist1;...;artistn;},dur_secs,bpm,extra
library_track_1=MP3,title,{artist1;...;artistn;},duration,bpm,bitrate,has_tags
library_track_2=WAV,title,{artist1;...;artistk;},duration,bpm,sample_rate,bit_depth
# Cache Settings
controller_cache_size=(integer)
```

```
# Mixing Settings
bpm_tolerance=(integer)
auto_sync={true, false}

# Playlists
# Format: playlistname= index_1, index_2,....,index_m
```

Important points:

- Artists are given inside braces and separated by semicolons: `\{Artist A;Artist B;\}`
- Playlists reference library tracks by numeric indices.
- Repeated indices are allowed (a track may appear multiple times in a playlist).

The config file will be located in the `bin` directory. You can assume that the configuration file is well-formed (i.e., it contains no syntax errors).

7.1.2 Real-World Parallel

Professional DJ controllers like Pioneer CDJ-3000 have:

- **USB/Network Storage** (Your Library): Thousands of tracks
- **Controller Memory**: Limited number of slots for quick access
- **Physical Decks**: Currently playing tracks

7.2 System Behavior

In order to run the system itself, you should launch the program with the `-I` flag to enable interactive mode, which allows you to select playlists and simulate a DJ session. More about this in the Q&A section in the Appendix. When launching the program with `-A` flag, the program runs in automatic mode, and will play all the playlists defined in the configuration file sequentially without user interaction.

At startup, the program reads `bin/dj_config.txt`. It parses the system's configuration and playlists from the configuration file, lists them, and prompts the user to select one interactively, re-prompting on invalid input (menu implementation provided). The selected playlist becomes a canonical Playlist object with AudioTrack instances. A fixed-capacity controller cache, defined in the config, stores tracks using an **LRU** policy keyed by library indexes. For each track occurrence in playlist order, if it's in the cache (HIT), it becomes most-recently-used; if not (MISS), it's inserted via polymorphic clone, and if full, the least-recently-used entry is evicted.

Playback uses two mixer decks (A and B), with instant transitions between them. For each new track in the playlist, if there is an empty deck, the track is loaded onto it; if both decks are occupied, the non-active deck is selected for replacement. A **polymorphic clone** of the track is taken from the cache without removing the cached copy. If the chosen deck is not empty, it is unloaded before loading the new track. The system then calls `load()` and `analyze_beatgrid()`

on the cloned track. Deck BPMs are compared: if the BPM difference between the two decks exceeds `bpm_tolerance` and `auto_sync` is true, the program updates the new track's BPM to match the average of the two and logs a message (no audio processing is performed). After loading and analysis, the newly loaded track becomes active, and the previously active deck is immediately unloaded.

If your system encounters errors, it displays a clear message and aborts the process. Unreadable lines or clone failures are logged and skipped. Ultimately, your program prints a summary of totals for tracks processed, cache hits, misses, evictions, loads per deck, transitions, and errors.

7.2.1 Layers and Responsibilities

1. Configuration & Parsing Layer (Infrastructure): This part handles reading configuration settings and parsing playlist files. It includes:

- `ConfigurationManager`: Handles all application settings
- `SessionFileParser`: Parses the config files

This part is already implemented for you.

2. Service Layer: This layer contains the core business logic organized into specialized services:

- `DJLibraryService`: Manages playlist operations and track library
- `DJControllerService`: Handles cache management with LRU eviction policy
- `MixingEngineService`: Manages deck operations and track analysis

About Service Architecture: Each service has a specific responsibility, making the system modular and easier to test. The controller service simulates real DJ hardware constraints with limited cache capacity.

3. Cache Management Layer: This layer manages limited memory capacity, simulating real DJ controller hardware constraints:

- `LRUCache`: Implements least-recently-used eviction algorithm
- `CacheSlot`: Represents individual cache entries with metadata

About LRU policy: When the cache reaches capacity, it evicts the least recently accessed track. This mirrors how real DJ controllers manage their limited internal memory.

4. Orchestration Layer: This layer coordinates the entire workflow:

- `DJSession`: Acts as the main orchestrator, coordinating between services and managing the overall application flow

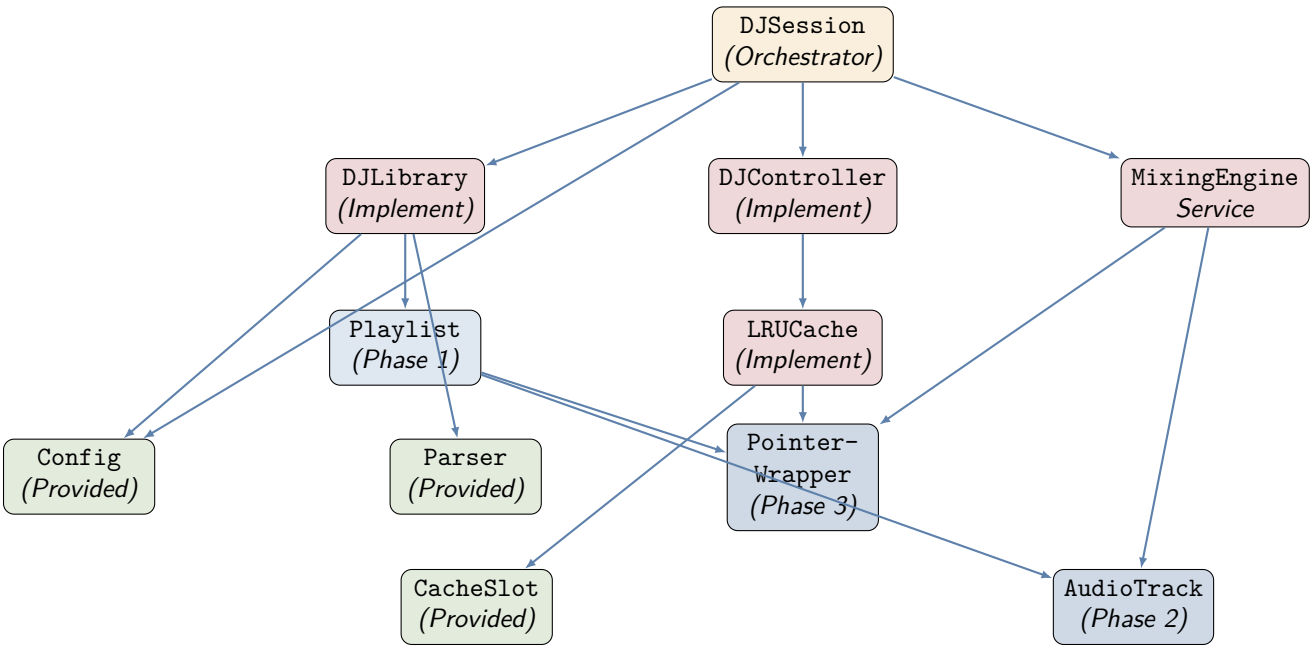
7.2.2 How Previous Phases Connect to Phase 4

Provided The `ConfigurationManager` & `SessionFileParser` are already implemented for you, so you can focus on the service and orchestration layers. The interactive menu system and configuration file parsing in `DJSession` are also provided. In addition, `CacheSlot` implementation for individual cache entries is also provided.

Required Given that you reached this phase, you should have already implemented a working `Playlist` class (Phase 1), a complete `AudioTrack` hierarchy with Rule of 5 (Phase 2), and the `PointerWrapper` class (Phase 3). These components are essential for the services you will implement in this phase.

In this phase In this phase, you will implement the three main services and the orchestrator. Each service will utilize your previously implemented classes to manage playlists, cache tracks, and simulate deck operations.

- `DJLibraryService`: Integrate your `Playlist` and `AudioTrack` classes for library management
- `DJControllerService`: Complete cache management using your `CacheSlot` and `PointerWrapper`
- `MixingEngineService`: Implement deck management and track analysis
- `DJSession`: Complete the orchestrator to coordinate all services



Phase 4 — Code Reading and System Understanding (not for submission)

Before continuing to read further ahead and before you begin implementing, carefully read the provided code skeleton for the following classes:

- `LRUCache` (`include/LRUCache.h`, `src/LRUCache.cpp`)
- `DJLibraryService`, `DJControllerService`, `MixingEngineService` (headers and sources)
- `DJSession` (header and source)

For each class, answer the following questions while reading further:

1. What is the main responsibility of this class?
2. How can you test this class in isolation?
3. Which previous-phase components does it depend on?
4. Is this a good starting point for implementation? Why or why not?
5. Which methods are already implemented, and which are marked as TODO?
6. How does this class interact with the other services and the orchestrator?
7. Sometimes, a service class uses another class as a helper (e.g., `LRUCache` in `DJControllerService`). What is the purpose of this helper class? What do you think the responsibility of the Service class is vs the helper class?
8. Sometimes, the logic required from a method might be complex. Can you identify any methods that might require multiple steps (i.e, utility methods) or careful handling of resources (e.g., cloning, loading, unloading)?
9. Do you need to override any member of the rule of 5 in any of these classes? Why or why not?

7.3 Implementation Guidance

The following sections break down the implementation steps for the LRU Cache, DJ Controller Service, and DJ Mixing Engine Service. It is up to you to decide the order in which you implement these components.

7.3.1 Implementation Steps for LRU Cache and DJ Controller Service

The `DJControllerService` simulates a professional DJ controller's memory management system, where tracks must be cached for quick access during live performance. Real controllers, such as the Pioneer CDJ-3000, have limited internal memory (typically 8-16 slots) that stores frequently accessed tracks to minimize loading delays. The service acts as an intermediary between the library (which owns the entire track collection and the playlists) and the mixing engine (which needs fast access to tracks for deck loading).

The underlying `LRUCache` implements a Least Recently Used (LRU) eviction policy with a fixed capacity of slots, as defined in the configuration file and mirroring hardware constraints. When the

cache reaches capacity, it automatically evicts the track that has been least recently accessed to make room for new ones. Each cache operation updates access timestamps, ensuring that actively used tracks remain available while older tracks are removed. The cache stores `AudioTrack` objects wrapped in your `PointerWrapper` from Phase 3.

Phase 4 — Implement LRU Cache and DJ Controller Service

1. **Understand the Cache Architecture**:
   - The `LRUCache` manages a fixed-capacity array of `CacheSlot` objects
   - Each slot stores: track pointer (via `PointerWrapper`), access timestamp, and occupied flag
   - Track identification is done via `track->get_title()` when needed
   - The cache uses an access counter that increments on every access to maintain LRU ordering
   - Cache keys are track titles (strings). Titles must be unique within a session.
2. **Implement LRU Cache Core Operations**:
   - **`findLRUSlot()`**: Iterate through all occupied slots to find the one with the minimum `last_access` value. Return that slot's index, or `max_size` if the cache is empty (no occupied slots).
     Example scenarios:
     - Cache state (slot: access_time): [0:100, 1:50, 2:empty] → returns 1 (lowest access time), [0:empty, 1:empty] → returns `max_size` (no occupied slots), [0:100, 1:200] → returns 0 (lower access time)
   - **`put(track)`**: Insert a cloned track object (wrapped in `PointerWrapper`). The method must:
     (a) Handle `nullptr` track by returning immediately
     (b) Check if a track with the same title already exists in the cache:
       - If exists: update its access time to make it most recently used and return `false` (no eviction occurred)
     (c) If the cache is full (all slots occupied), call `evictLRU()` first
     (d) Find an empty slot using `findEmptySlot()`
     (e) Store the new track with the current `access_counter` value and mark the slot as occupied
     (f) Return `true` if an eviction occurred, `false` otherwise

     Example scenarios:
     - Empty cache (size 2): put(A) → no eviction, returns `false`
     - Cache [A]: put(B) → no eviction, returns `false`, Cache [A,B]: put(C) → evicts LRU, returns `true`, Cache [A,B]: put(A) → updates A's access time, returns `false` (no insertion)
3. **Implement DJ Controller Service**:

- **`loadTrackToCache(source_track)`**: This method manages cache population with the following workflow:
  (a) Check if track with the same title exists in cache using `cache.contains(track.get_title())`
  (b) **HIT case**: If found, call `cache.get()` to refresh MRU status and return 0 `1`
  (c) **MISS case**: If not found, proceed with insertion:
    - Create a polymorphic clone of the track using and unwrap the `PointerWrapper` to get the raw pointer
    - If clone is `nullptr`, log error and return (or appropriate failure code)
    - Simulate loading on the cloned track, and do a beatgrid analysis.
    - Wrap the prepared clone in a new `PointerWrapper` and insert into cache using the appropriate method
    - If `put()` returns `true` (eviction occurred), return `-1`
    - If `put()` returns `false` (no eviction), return `0`
    - return 1 on HIT (track found and refreshed), 0 on MISS without eviction (track inserted), -1 on MISS with eviction (LRU track evicted)
- **`getTrackFromCache(track_title)`**: Look up track in cache by title using `cache.get(track_title)`. Return the pointer if found, `nullptr` otherwise. This method does not transfer ownership.
- Handle clone failures gracefully by logging `[ERROR] Track: "<title>" failed to clone` and returning appropriate failure code without corrupting cache state

4. **Integration with Library Service**:
   - Cache keys are *titles*. Ensure the library enforces unique titles or documents the constraint.
   - Use polymorphic clone to create independent copies for cache ownership
   - Ensure proper cleanup when cache evicts tracks.
   - Maintain clear ownership: library owns canonical tracks, cache owns cloned copies

Real-World Connection

Professional DJ controllers, such as the Pioneer CDJ-3000, typically have limited internal memory (8-16 track slots) and must provide instant track access during live performances. Your LRU cache implementation simulates this hardware constraint, where efficient caching and eviction directly impact the DJ's ability to seamlessly mix tracks without audio dropouts or loading delays. The cache acts as a "hot storage" layer between the full library and the active mixing decks.

For submission: Complete `LRUCache.cpp` and `DJControllerService.cpp` with all TODO methods implemented, demonstrating proper LRU eviction policy, polymorphic track cloning, memory management via `PointerWrapper`, and accurate statistics tracking.

7.3.2 Implementation Steps for the DJ Mixing Engine Service

The `MixingEngineService` simulates professional DJ mixing hardware with two physical decks (A and B) that manage track playback and transitions. Real DJ mixers, such as the Pioneer DJM-900NXS2, handle instant deck switching, automatic BPM synchronization, and seamless crossfading between tracks. The service manages the lifecycle of tracks on each deck, including loading, analysis, playback, and unloading, when tracks are finished or replaced.

The service operates on an instant-transition policy, maintaining two decks where only one is active at a time. When loading a new track, the system automatically targets the non-active deck, performs track preparation (cloning, loading, and analysis), and immediately switches to make it active while unloading the previously active deck. This creates seamless transitions, where each new track instantly replaces the previous one. The service also handles BPM compatibility checking and automatic synchronization based on tolerance settings, mirroring real-world DJ workflow requirements.

Phase 4 — Implement Mixing Engine Service

1. **Understand the Service Architecture**:
   - The service uses a two-deck array (`decks[0]` and `decks[1]`) with raw pointers
   - An `active_deck` index (0 or 1) tracks which deck is currently playing
   - Configuration parameters `auto_sync` and `bpm_tolerance` control mixing behavior
   - The instant-transition policy ensures only one deck is loaded after each operation
2. **Implement Constructor and Destructor**:
   - **Constructor**: Initialize both deck pointers to `nullptr`, set `active_deck` to 0, initialize `auto_sync` to `false`, and `bpm_tolerance` to 0. Log initialization message: `[MixingEngineService] Initialized with 2 empty decks.`
   - **Destructor**: Log cleanup message: `[MixingEngineService] Cleaning up decks...`. Iterate through both decks, delete any non-null tracks, and set pointers to `nullptr`. Ensure no memory leaks during shutdown.
3. **Implement Core Loading Workflow (`loadTrackToDeck`)**:
   - **Initial State:** Both decks start empty (nullptr)
   - **First Track:** Load to deck 0, set as active, leave deck 1 empty
   - **Subsequent Tracks - Instant Transition Pattern:**
     (a) Log section header: `\n=== Loading Track to Deck ===`
     (b) Clone track polymorphically using and wrap in `PointerWrapper` for safety.
     (c) If clone fails, log error and return -1: `[ERROR] Track: "<title>" failed to clone`
     (d) Identify target deck: Calculate as `1 - active_deck` (the one that is NOT currently active)
     (e) Log deck switch: `[Deck Switch] Target deck: <index>`
     (f) Unload target deck if occupied:
       – Check if `decks[target]` is not `nullptr`, and delete the track and set

pointer to `nullptr`

(g) Perform track preparation on cloned track by simulating loading and beat analysis.

(h) **BPM Management:** If an active deck exists and `auto_sync` is enabled, if the BPM difference exceeds tolerance (i.e, can't mix with the cloned track), synchronize the BPM on the cloned track.

(i) Release pointer from `PointerWrapper` and assign to `decks[target]` and log: `[Load Complete] '<title>' is now loaded on deck <target>`

(j) **Instant Transition:** Immediately unload the previously active deck:

- If this is not the first track (`active_deck` has a loaded track)
- Log unload: `[Unload] Unloading previous deck <active_deck> (<title>)`
- Delete the track from `decks[active_deck]` and set to `nullptr`

(k) Switch active deck: Set `active_deck = target`

(l) Log active deck switch: `[Active Deck] Switched to deck <target>`

(m) Return target deck index for statistics tracking

- **Error Handling:** If clone fails, return -1 without modifying deck state
- **Final State:** Only the newly loaded track remains on its deck; other deck is empty

4. **Implement BPM Compatibility Check (`can_mix_tracks`)**:
   - Verify that `decks[active_deck]` is not `nullptr` (return `false` if empty)
   - Verify that the input track wrapper contains a valid pointer (return `false` if null)
   - Get BPM values from both active deck and new track using `get_bpm()`
   - Calculate absolute BPM difference and return `true` if difference is less than or equal to `bpm_tolerance`, otherwise `false`
5. **Implement BPM Synchronization (`sync_bpm`)**:
   - Verify both active deck and new track are valid (non-null)
   - Get original BPM from the new track for logging
   - Calculate average BPM and update the new track's BPM.
   - Log synchronization: `[Sync BPM] Syncing BPM from <original> to <new>`
6. **Memory Management Requirements**:
   - Use `PointerWrapper` during cloning to ensure exception safety
   - Release ownership from `PointerWrapper` using `release()` only after track is fully prepared
   - Ensure proper cleanup during deck unloading: delete old track before assigning a new one
   - Destructor must clean up any remaining tracks on both decks
   - Ensure no memory leaks during repeated load/unload cycles

Real-World Connection

Professional DJ software enables instant transitions to create seamless mixes, where new tracks seamlessly replace old ones. Your implementation simulates this critical workflow where timing precision and memory safety are essential. The automatic BPM synchronization feature mirrors real DJ controllers, helping to maintain smooth tempo transitions and prevent jarring speed changes that would disrupt the dance floor.

For submission: Complete `MixingEngineService.cpp` with all TODO methods implemented, demonstrating proper instant-transition deck management, BPM synchronization, polymorphic cloning via `PointerWrapper`, and leak-free memory handling throughout the track lifecycle.

7.3.3 Implementation Steps for DJ Library Service

The `DJLibraryService` serves as the central repository manager for the DJ system, much like professional software such as Serato DJ or rekordbox manages vast music collections. The service maintains canonical ownership of all track data throughout the session, serving as the authoritative source for track metadata and ensuring consistent track identity across the system. It bridges the gap between the unified configuration file (which defines the entire library and playlists) and the structured track objects needed by the cache and mixer components.

The service handles the complete track lifecycle from configuration parsing to object creation. Use `SessionFileParser::parse_config_file` to produce a `SessionConfig` that contains *library_tracks* and a *playlists* map. Construct polymorphic `AudioTrack` objects (MP3 or WAV) based on format detection.

Phase 4 — Implement DJ Library Service

1. **Understand the Service Architecture**:
   - The service manages track data and metadata for the entire session.
   - It uses a collection of `AudioTrack` objects, indexed for quick access.
   - The service owns a `Playlist` object that references tracks via clones
   - Configuration parameters define the library structure and playlists.
2. **Implement Core Library Operations**:
   - **`buildLibrary(library_tracks)`**: Create canonical `AudioTrack` objects from `SessionConfig::library_tracks`. For each track:
     (a) Check the `format` field ("MP3" or "WAV")
     (b) Create appropriate track type using the provided metadata (check the config file layout).
     (c) Store the raw pointer in the `library` vector

(d) Log creation message:

- MP3: `MP3Track created: <bitrate> kbps`
- WAV: `WAVTrack created: <sample_rate>Hz/<bit_depth>bit`

Log summary: `[INFO] Track library built: <count> tracks loaded`

- **`loadPlaylistFromIndices(name, indices)`**: Build a `Playlist` from track indices:

(a) Log: `[INFO] Loading playlist: <name>`
(b) Create new `Playlist` with the given name
(c) For each index in the indices vector (1-based):

- Validate index is within library bounds.
- If invalid, log warning: `[WARNING] Invalid track index: <index>` and skip, if valid, get track from library using 0-based indexing: `library[index-1]`
- Clone the track polymorphically and unwrap the `PointerWrapper`.
- If clone is `nullptr`, log error and skip
- Call `load()` and `analyze_beatgrid()` on cloned track
- Add cloned track to playlist using `playlist.add_track()`
- Log: `Added '<title>' to playlist '<playlist_name>'`

Log summary: `[INFO] Playlist loaded: <name> (<count> tracks)`

- **`findTrack(title)`**: Search playlist for track by title:
  - Use `playlist.find_track(title)` to locate the track
  - Return pointer if found, `nullptr` otherwise
  - This provides a lookup without transferring ownership
- **`getTrackTitles()`**: Return vector of all track titles in current playlist:
  - Iterate through playlist tracks
  - Collect titles using `track->get_title()`
  - Return `std::vector<std::string>` of titles

3. **Integration with Configuration Parsing**:
   - The `DJSession` orchestrator calls `SessionFileParser::parse_config_file` to populate `SessionConfig`
   - Your library service receives parsed config via `buildLibrary(config.library_tracks)`
   - Artists are provided as a vector (already parsed from `{artist1;artist2;...}` format)
   - Track indices in playlists are 1-based (convert to 0-based for vector access)
   - Handle missing or invalid track indices gracefully (log warning and skip)
4. **Memory Management Requirements**:
   - Library owns canonical track objects stored in `library` vector

- Destructor must delete all tracks in `library` vector and clear it
- Playlist owns cloned copies of tracks (managed via `Playlist` destructor)
- Use `PointerWrapper` during cloning for exception safety
- Ensure no memory leaks when the service is destroyed

Real-World Connection

Professional DJ software maintains extensive music libraries with thousands of tracks loaded at startup. Your library service simulates this challenge by pre-loading all track metadata from a single configuration file, ensuring tracks remain instantly accessible for any playlist while maintaining consistent metadata and supporting efficient index-based lookup that DJs rely on during live performance.

For submission: Complete `DJLibraryService.cpp` with all TODO methods implemented, demonstrating proper library loading from configuration, index-based track retrieval, playlist construction, and integration with your Phase 1-3 components.

7.3.4 Implementation Steps for DJ Session Orchestrator

The inner loop iterates over each track in the selected playlist. The DJSession class serves as the central orchestrator for the entire DJ system, coordinating between all service layers to simulate a complete DJ performance workflow. Similar to how professional DJ software like Serato DJ or rekordbox manages session state, the orchestrator handles configuration loading, playlist selection, and the main playback loop that processes tracks in sequence. It maintains the overall system state while delegating specific responsibilities to specialized services: library management, cache operations, and mixing engine control.

The session follows a structured workflow that mirrors real DJ performance: startup configuration parsing, interactive playlist selection from available files, and then sequential track processing with instant transitions between decks. Each track in the playlist triggers a complete workflow: cache management (hit/miss/eviction), polymorphic cloning for deck ownership, format-specific loading and analysis, BPM compatibility checking, and automatic deck switching. The orchestrator maintains comprehensive statistics throughout the session, tracking cache performance, deck utilization, transition counts, and error handling, providing detailed reporting that helps understand system behavior and performance characteristics.

The class operates in two different modes: Interactive mode (default) and All-Playlists mode (if the `-A` flag is also provided). In interactive mode, the user selects which playlist to play in a loop until they choose to quit. In all-playlists mode, the system iterates through all available playlists in sorted order, processing each in turn until all are completed.

Phase 4 — Implement DJ Session Orchestrator

1. **Understand Provided Components**:
   - **`load_configuration()`**: Already implemented - parses `bin/dj_config.txt`,

initializes service settings
- **`display_playlist_menu_from_config()`**: Already implemented - shows sorted playlist menu, validates input
- **`print_session_summary()`**: Already implemented - displays final statistics
- **`load_playlist(playlist_name)`**: Already implemented - loads playlist from config indices into library service

2. **Implement `load_track_to_controller(track_name)`**:
   (a) Find track in library using `library_service.findTrack(track_name)`
   (b) If track not found:
      - Log error: `[ERROR] Track: "<track_name>" not found in library`
      - Increment `stats.errors`
      - Return `0`
   (c) Log: `[System] Loading track '<track_name>' to controller...`
   (d) Call `controller_service.loadTrackToCache(*track)` (pass by reference)
   (e) Interpret return value according to contract:
      - `1`: Cache HIT - Increment `stats.cache_hits`
      - `0`: Cache MISS (no eviction) - Increment `stats.cache_misses`
      - `-1`: Cache MISS with eviction - Increment both `stats.cache_misses` and `stats.cache_evictions`
   (f) Return the cache result code for the caller's use
3. **Implement `load_track_to_mixer_deck(track_title)`**:
   (a) Retrieve track from cache using the appropriate method in the controller service.
   (b) If track not in cache:
      - Log error: `[ERROR] Track: "<track_title>" not found in cache`
      - Increment `stats.errors` and return `false`
   (c) Load the track to the deck using the service and handle the returned value
      - If `0`: Increment `stats.deck_loads_a` and `stats.transitions`
      - If `1`: Increment `stats.deck_loads_b` and `stats.transitions`
      - If `-1`: Log error and increment `stats.errors`, return `false`
   (d) Return `true` on successful deck load (deck index 0 or 1)
4. **Implement `simulate_dj_performance()` Main Loop**:
   (a) Configuration and initialization are already handled (see provided code in skeleton)
   (b) **Playlist Selection Loop**:
      - If `play_all` is `true`:
        – Extract and sort all playlist names from `session_config.playlists` and iterate through each playlist name
      - If `play_all` is `false` (interactive mode):

- Call `display_playlist_menu_from_config()` to get user selection, if an empty string is returned (cancelled), break the loop

(c) **For Each Selected Playlist**:

i. Call `load_playlist(playlist_name)`

ii. If load fails, log error and continue to next playlist (or prompt again in interactive mode)

iii. **Track Processing Loop** - for each track in `track_titles`:

- Log: `\n-- Processing: <track_title> --`
- Increment `stats.tracks_processed`
- **Cache Loading Phase**:
  - load the track to controller using `load_track_to_controller(track_title)`
  - update the cache statistics based on the return value
- **Deck Loading Phase**:
  - Call `load_track_to_mixer_deck(track_title)` and update the deck and transition statistics based on the return value
  - If load fails, continue to next track (error already logged and counted)

iv. After all tracks processed, call `print_session_summary()`

v. Reset statistics for next playlist: set all `stats` members to 0

(d) **Loop Continuation**:

- In `play_all` mode: continue until all playlists processed, in interactive mode: continue until user selects Cancel (0)

(e) After loop completion, log: `Session cancelled by user or all playlists played.`

5. **Statistics Tracking Requirements**:
   - **Cache misses**: Count both with and without eviction (a MISS with eviction increments both `cache_misses` and `cache_evictions`)
   - **Transitions**: Increment whenever a track successfully loads to either deck
   - **Errors**: Track failures at any stage (library lookup, cache access, deck loading, clone failures)
   - **Tracks processed**: Total tracks attempted (regardless of success/failure)
6. **Error Handling Requirements**:
   - Abort early if configuration parsing fails or no playlists are found
   - Log all errors with format: `[ERROR] <Context>: "<identifier>" <description>`
   - Skip individual track failures gracefully without stopping the session
   - Continue processing remaining tracks even after errors
   - Ensure all errors are counted in `stats.errors`

Real-World Connection

Professional DJ software must orchestrate complex workflows seamlessly, managing everything from library scanning to real-time mixing operations. Your session orchestrator simulates the critical coordination challenges that DJ software faces, where any failure in the workflow can disrupt a live performance and disappoint audiences.

For submission: Complete `DJSession.cpp` with all TODO methods implemented, demonstrating proper service orchestration, comprehensive error handling, statistical tracking, and integration with all Phase 1-4 components.

7.4 Integration Questions

Phase 4 Integration Questions — not for submission

Memory Management and Ownership:

1. Do `DJSession`, `DJLibraryService`, `DJControllerService`, or `MixingEngineService` need custom Rule of 5 implementations? Why or why not?
2. Explain the ownership boundaries in the system: Who owns the canonical tracks? Who owns cached copies? Who owns deck-loaded tracks?
3. Why clone tracks instead of moving them from cache to mixer? What would go wrong if we moved instead?
4. What happens to mixer memory when `DJSession` is destroyed? Trace the destruction order.
5. How does `PointerWrapper` help prevent memory leaks during track transitions between layers?

System Architecture and Integration:

6. How does the LRU algorithm decide which track to evict when the cache is full? Walk through a scenario with 3 tracks and a capacity of 2.
7. Why doesn't `MixingEngineService` store playlists directly? What design principle does this demonstrate?
8. Explain how polymorphic cloning works across the cache-to-mixer boundary. What happens if the clone fails?
9. What are the performance implications of the three-tier memory hierarchy (Library $\rightarrow$ Cache $\rightarrow$ Mixer)?
10. How do the services communicate? What data flows between `DJSession`, `DJLibraryService`, `DJControllerService`, and `MixingEngineService`?

Error Handling and Edge Cases:

11. How does the system handle file I/O failures during configuration loading?

12. What happens if a track title in the playlist doesn't exist in the library?
13. How does the system respond to memory pressure when the cache is full?
14. What occurs if BPM synchronization fails or produces an invalid BPM value?
15. Trace what happens when loading a track to a deck fails mid-operation. How is the system state preserved?

8 Final Integration Test

Once you've completed all phases, run the end-to-end checks to validate the complete system. The goals here are: a clean build, zero leaks, correct cache/mixer behavior, and interactive selection working with real files.

```
# Clean build
make clean

# Optional: Run with valgrind if available
make test-leaks

make clean
# build the final version:
make release

#run the program
./bin/dj_manager -I

make clean
```

Your complete system should demonstrate:

- **Compilation**: No warnings or errors in any phase
- **Memory Safety**: Zero memory leaks detected by AddressSanitizer and valgrind
- **File Integration**: Successfully parse configuration from `bin/dj_config.txt`.
- **Cache Management**: Efficient LRU eviction and track deduplication
- **Rule of 5 Mastery**: Working copy/move semantics in complex scenarios
- **Exception Safety**: Robust error handling during file I/O and resource allocation
- **System Integration**: Seamless interaction between all Phase 1-4 components
- **Performance**: Reasonable memory usage even with large playlists

8.1 Expected Test Output

Under testing mode (i.e, without `-I` flag), the output should look exactly like in the `test_output.txt` file located in the root directory. In interactive mode (with the `-I` flag), the output will vary based on user input. The output should resemble exactly what is in the `interactive_output.txt` file located in the root directory, assuming the user selects the first playlist.

9 Grading

As mentioned in class, your submission will be evaluated during a review session. The testing procedure will be covered by our testing assistants in CS labs or via an online meeting if you are

on reserve duty. The sessions will start after the submission deadline, and you will be notified of your scheduled time via email. We will also explain it when it is close to the deadline. During the session, the tester will review your code using a set of predefined tests and will ask you questions about your implementation.

9.1 Oral Examination Focus Areas

During the oral examination, you will be asked to:

- **Explain design decisions**: Why did you choose specific ownership models or data structures?
- **Demonstrate understanding**: Walk through complex scenarios like memory pressure or exception handling
- **Trace object lifetimes**: Show how objects flow through the system from creation to destruction
- **Justify trade-offs**: Discuss performance vs. memory usage decisions in your implementations
- **Handle edge cases**: Explain how your code responds to error conditions and resource constraints
- **Class Material Connections**: Relate your implementation to concepts covered in lectures and readings, as well as questions about C++ concepts.
- **Code Walkthrough**: Be prepared to navigate your codebase and explain key functions and classes

10 Conclusion

Congratulations on bringing all phases together into a leak-free, well-structured system. You now own the core skills: disciplined ownership and RAII, Rule of 5 in a polymorphic hierarchy, a minimal smart-pointer wrapper, and a multi-service orchestration that respects memory boundaries. These patterns will carry into any systems project you tackle next. Well done!

A Appendix: Environment Setup

Note: If you already configured your development environment using the instructions provided in the course's Moodle website, you can skip this section. To effectively complete assignments in this course, you'll need a Linux environment. We'll utilize Docker with the Windows Subsystem for Linux (WSL) backend to compile and run code within a development container, facilitated by the Dev Containers extension for Visual Studio Code (VS Code).

This guide is comprised of two main parts: one for a one-time setup and the other for a per-assignment setup.

1. Installing the required tools and extensions:
   - Install WSL 2
   - Install Docker Desktop
   - Install VS Code
   - Install the Dev Containers extension for VS Code
2. Configuring the development environment for each new assignment.

A.1 Installation Steps

Before you begin:

- **Administrative Privileges:** Ensure you have the necessary permissions to install and run Docker, especially if you're using a workplace-provided computer.
- **Enable Virtualization:** (Windows and Linux users only) Verify that virtualization is enabled in your system's BIOS settings. For guidance, refer to Microsoft's instructions on enabling virtualization on Windows.

Windows Subsystem for Linux (WSL) - Windows Users Only

WSL, also known as the Windows Subsystem for Linux, allows you to run a Linux distribution alongside your Windows installation, providing a native Linux experience. We'll use WSL as the backend for Docker to run Linux containers.

1. Open PowerShell or Command Prompt as Administrator:
   Right-click on the Start button and select "Windows PowerShell (Admin)" or "Command Prompt (Admin)".
2. Install WSL by executing the following command:

   ```
   wsl --install
   ```

   This command installs WSL and the default Linux distribution (usually Ubuntu).
3. Restart Your Computer:
   After installation, restart your computer to apply the changes.

Docker Desktop

Docker is an application for building, sharing, and running containerized applications and microservices. In a sense, containers virtualize runtimes and libraries needed for an application.

Installation Steps:

- Windows users:
  Download and install Docker Desktop for Windows.
- Linux users:
  - Ubuntu-based Distributions:
    Follow the installation instructions for Docker Desktop on Ubuntu.
  - Fedora:
    Refer to the Fedora-specific Docker Desktop installation guide.
- macOS users:
  Download and install Docker Desktop for macOS.
  An alternative to Docker Desktop on macOS is colima, a lightweight alternative that sometimes works better. Try it if you encounter issues with Docker Desktop.

Visual Studio Code and Dev Containers

VS Code, developed by Microsoft, is a versatile code editor that supports a wide range of programming languages. It has an extension called Dev Containers, allowing a container to be a full-featured development environment. Using the configuration files we provide as part of the assignments, you can easily create new environments for each assignment in this course. Once you have installed everything successfully, you can connect to the development environment by pressing the "Reopen in Container" button in VS Code.

Let's break it down:

1. Install VS Code:
   Download and install Visual Studio Code for your operating system.
2. Install the Dev Containers Extension:
   - Click on the Extensions icon in the sidebar or press ctrl+shift+X (or cmd+shift+X on macOS).
   - In the search bar, type "Dev Containers".
   - Locate the "Dev Containers" extension by Microsoft and click "Install".
3. After installation, you should see an indicator in the bottom-left corner of VS Code, confirming the Dev Containers extension is active.

A.2 Configuring the Development Environment

Set Up the .devcontainer Directory

We have included a `.devcontainer` directory in the root directory for this assignment's root that contains the necessary configuration files for the dev container. To set up the environment

for this assignment, follow these steps:

1. Clone or copy the assignment repository to your local machine, creating a new directory for each assignment.
2. Open the project in VS Code by selecting the folder that contains the `.devcontainer` directory.
3. Upon opening the project, you should see a prompt to "Reopen in Container" in the bottom-right corner. Click this button.
4. VS Code will reload and begin setting up the development container. This process may take a few minutes during the initial setup.
5. If you are not prompted, press `ctrl+shift+P` (or `cmd+shift+P` on macOS) to open the Command Palette. Type "Dev Containers: Reopen in Container" and select it from the list.
6. When finished, VS Code will show a message indicating that it is working on a container. You can click on "Show Log" to view detailed progress and troubleshoot any potential issues.
7. Once the setup is complete, you should see an indication in the status bar confirming that you're connected to the development container.
8. Additionally, opening a new terminal in VS Code should display a prompt indicating that you're operating within the container environment.

B Appendix: Additional Resources

- C++ Rule of Three/Rule of Five
- C++ Move Semantics
- RAII (Resource Acquisition Is Initialization)
- Smart Pointers
- Valgrind
- SOLID principles in C++
- C++ Operators
- C++ Exception Handling

C Appendix: FAQ

- **I need an extension for the deadline, how can I get one?** Please refer to the course policies on Moodle regarding extensions. Extensions are granted only under exceptional circumstances and must be requested before the deadline. Every request should be sent to Hedi via email.
- **Can I use external libraries for this assignment?** No, you are not allowed to use any external libraries. You must implement all required functionality using only the C++ standard library and the provided codebase.
- **What should I do if I encounter a bug in the provided codebase?** If you find a bug in the provided code, please report it to the relevant teaching assistants as soon as possible. Include a detailed description of the issue and any steps to reproduce it.

- **Can I discuss the assignment with my classmates?** You are encouraged to discuss general concepts and approaches with your classmates. However, you must write your own code and refrain from sharing or copying code from others. others. Plagiarism will result in severe penalties as outlined in the course policies.
- **How can I get help if I'm stuck?** You can seek help for the assignment from the teaching assistants who are assigned to this assignment during office hours or via the course discussion forum. Be sure to provide specific details about the issue you're facing to get the most effective assistance. You can also refer to the course materials and resources for further guidance.
- **Can I join an existing pair?** The assignments are meant to be completed in pairs of 2 students. If you do not have a partner, please find a partner among your classmates or contact the teaching assistants for assistance. If you wish to join an existing pair, you must obtain approval from Hedi via email before doing so.
- **What is the principle of separation of concerns? Separation of concerns** is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program. The goal of separation of concerns is to establish a well-organized codebase where each module or component has a clear responsibility, making it easier to maintain, understand, and modify.
- **What is `std::move`?** `std::move` is a standard library function that casts an object to an rvalue reference, enabling move semantics. It does not actually move anything; instead, it indicates that the object can be "moved from," allowing resources to be transferred rather than copied.
- **What is a command-line argument?**
- A command-line argument is a parameter passed to a program when it is executed from the command line or terminal. These arguments can modify the program's behavior, such as specifying input files, configuration options, or operating modes. In C++, command-line arguments are typically accessed through the parameters of the `main` function: `int main(int argc, char* argv[])`. For example, in the command `./program -I`, `-I` is a command-line argument that might indicate interactive mode.