MMC 7

Sunday, May 11, 2025 11:49 AM

1. Abstraction Levels

Syllabus Topic:

- Abstraction level of the programming of multimedia systems

Related Past Questions:

- Explain the abstraction levels of the programming in brief.
- Explain the abstraction levels of programming use in a multimedia system with a diagram.
- Explain the abstraction levels of programming.

2. Libraries

Syllabus Topic:

- Introduction to Libraries

Related Past Questions:

(No direct PYQs found for this topic)

3. System Software

Syllabus Topic:

- Data as Time capsules
- Data as Streams

Related Past Questions:

(No direct PYQs found for this topic)

4. Toolkits

Syllabus Topic:

- Introduction to toolkits

Related Past Questions:

(No direct PYQs found for this topic)

5. Higher Programming Languages

Syllabus Topic:

- Media as types
- Media as files
- Media as processes
- Programming language requirements
- Interprocess communication mechanism
- Language

Related Past Questions:

- How can you define media as type, media as file, and media as process in higher programming languages?

6. Object-Oriented Approaches

Syllabus Topic:

- Class, object, inheritance, polymorphism
- Application-specific metaphors as classes
- Application-generic metaphors as classes
- Devices as classes
- Processing units as classes
- Distribution of BMOs and CMOs
- Media as classes
- Communication-specific metaphors as classes

Related Past Questions:

- Explain the abstraction levels of programming using object-oriented approaches.
- What are Basic Multimedia Objects (BMO) and Compound Multimedia Objects (CMO)? How can you define BMO

and CMO using object-oriented approach?
- Explain the abstraction levels of programming using object-oriented approaches.

Abstraction Levels in Multimedia System Programming

What is Abstraction?

- **Abstraction** is the process of hiding internal details and showing only the necessary parts to the user or programmer.
- In **multimedia systems**, abstraction is used to **simplify programming**, manage **media content**, and support **real-time interaction**.
  Example:
  When you click "Play" on a video player, you don't need to know how the data is decoded or how audio is synchronized. The underlying complexity is abstracted.

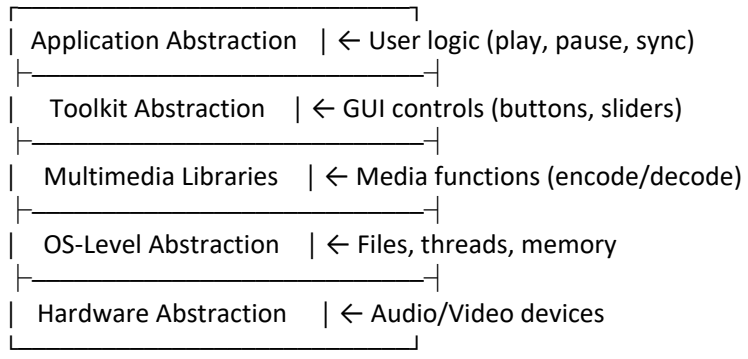
Why is Abstraction Important in Multimedia?

- Multimedia systems involve multiple media (text, images, audio, video) that must be **synchronized**.
- It reduces **complexity** by dividing the system into **layers**.
- Improves **code reuse**, **portability**, and **maintainability**.

Levels of Abstraction in Multimedia Programming

Level	What it Does	Example
1. Hardware Abstraction	Hides device-specific details. Communicates with hardware via drivers/APIs.	DirectX, OpenGL, OpenAL hide GPU/audio card details
2. OS Abstraction	Manages files, memory, threads, I/O for multimedia operations	Windows/Linux API for file access, threads, real-time scheduling
3. Library Abstraction	Provides reusable media functions like encode(), play(), compress()	FFmpeg, SDL, OpenCV
4. Toolkit Abstraction	Handles GUI elements and user interaction	Qt, JavaFX, GTK: used for sliders, buttons, timeline bars in media apps
5. Application Abstraction	Contains the app-specific logic and interactions between media components	VLC player logic: sync audio with video, volume normalization, playlist

Simple Layered Diagram



Example: Playing a Video File

Abstraction Level	What Happens in the Video Player Example
1. Hardware Abstraction	Your app doesn't directly talk to GPU, sound card, or storage — it uses drivers (e.g., sound card driver).
2. OS Abstraction	The player asks the operating system to open the file, allocate memory, and manage threads for audio/video.
3. Library Abstraction	It uses FFmpeg to decode the video and audio streams instead of writing decoding logic from scratch.
4. Toolkit Abstraction	The GUI (buttons like play, pause, slider, volume) is made using a toolkit like Qt or JavaFX.
5. Application Abstraction	The app logic controls video sync, handles playlists, remembers where the user stopped watching, etc.

Benefits of Abstraction

- Simplifies complex system development
- Improves **modularity** and **reusability**
- Makes **maintenance** and **debugging** easier
- Allows focus on **high-level logic** without worrying about low-level details

Libraries

Definition:
A **library** is a collection of **pre-written code** that can be reused in different programs. It provides **functions and classes** that help in building multimedia systems without writing everything from scratch.

Use in Multimedia Systems:

- **Saves time**: Common multimedia functions (like playing audio, resizing images) are already available.
- **Standardized**: Libraries ensure consistent performance across applications.
- **Examples**:
    - OpenCV – for image/video processing.
    - FFmpeg – for audio/video compression and conversion.
    - SDL – for handling graphics, sound, and input devices.

System Software in Multimedia Programming

System software manages and controls multimedia devices and processes, often by integrating device access **directly into the operating system**, rather than through separate libraries.

Data as Time Capsules & Data as Streams

1. Data as Time Capsules:

- **Concept**: In multimedia, **time capsules** refer to files that carry both **data** and its **valid lifespan**.
    - These files store the **time** period during which the data (like video or audio) is valid or accessible.
    - Widely used in **video** more than in audio.

- **Example**:
  - A **video file** has a time capsule with its **start and end times**, defining when the content is valid.
  - This helps in managing video files, where frames may be stored with timestamps to ensure synchronization.

2. Data as Streams:

- **Concept**: A **stream** represents a continuous flow of data (e.g., audio/video), where data is processed **sequentially**.
  - Streams allow actions like **play, fast forward, rewind, and stop** on continuous media.
- **Example**:
  - **Audio stream** in a **music player**: The data flows continuously from the source (server) to the player and can be controlled via the **play**, **pause**, and **skip** functions.
- **Microsoft Windows MCI**:
  - Provides an interface to manage and control **media streams**, enabling access to and manipulation of continuous media, such as audio and video.

Toolkits

Definition:
A **toolkit** is a collection of pre-written software components that help developers build multimedia applications by providing ready-to-use functions for tasks like graphics, audio, and video handling.

Key Points:

- **Pre-built Components:** Simplifies tasks like audio/video playback and graphics rendering.
- **Cross-Platform:** Works across different platforms (Windows, macOS, Linux).
- **Abstraction:** Handles complex operations (e.g., media processing) with simple API calls.

Examples:

1. **SDL (Simple DirectMedia Layer):** Used for game development and multimedia applications.
2. **GStreamer:** Used for streaming media and real-time video processing.
3. **OpenCV:** Used for image and video processing, especially in computer vision applications.

Higher Programming Languages in Multimedia Systems

In the context of multimedia systems, **higher programming languages (HLLs)** are used to process continuous media data like audio, video, and text. These languages simplify the interaction with multimedia data and hardware, often abstracting hardware details and device driver interactions. Here's how media is handled in HLLs:

1. Media as Types

- **Definition:** In multimedia computing, media can be represented as **data types** in a programming language. This means that multimedia data like audio, video, or images can be treated as specific types, much like integers or characters in standard programming.
  - **Example:** In **OCCAM-2** (a language designed for parallel processing), you can define media as types, such as audio data types that can be processed by the program. For instance:
    - ldu.left1, ldu.left2 can represent left audio channels.
    - ldu.left_mixed might be a mixed audio output, where the program can perform operations like adding or multiplying the audio streams.
  - This allows easy manipulation and processing of media (like mixing two audio files), where the media stream is directly represented as a type.

2. Media as Files

- **Definition:** Another way to handle multimedia data is by treating it as **files**. Continuous media streams can be accessed like regular files, where the system reads, writes, and processes the media data using standard file

operations.

- **Example:** The media data can be treated like files:
  file_h1 = open(MICROPHONE_1)
  file_h2 = open(MICROPHONE_2)
  file_h3 = open(SPEAKER)
  read(file_h1)
  read(file_h2)
  mix(file_h3, file_h1, file_h2)
  activate(file_h1, file_h2, file_h3)
- Here, audio data streams are opened from the microphone files (MICROPHONE_1, MICROPHONE_2), processed (e.g., mixed together), and then outputted to the speaker file (SPEAKER).
- This abstraction simplifies media handling by leveraging familiar file management techniques.

3. Media as Processes

- **Definition:** In some programming languages, media can be represented as **processes**. This means that multimedia data handling is done through active processes that communicate and synchronize in real-time.
  - **Example:** The program creates processes (e.g., PROCESS_1, PROCESS_2) that handle different media streams concurrently. These processes can be scheduled and synchronized, allowing real-time operations such as simultaneous audio capture and playback.

4. Programming Language Requirements

- **Interprocess Communication (IPC) Mechanism:**
  - **Timing and Synchronization:** The programming language must support an **IPC mechanism** that handles the transmission of continuous media data like audio and video in a **timely** manner.
    - **Timely transmission** means that data must be delivered and processed at the right time, crucial for real-time applications.
    - **Synchronization** ensures that different processes that handle different media streams (audio, video) stay in sync, maintaining smooth playback or recording.
  - **Example:** An IPC mechanism must allow one process to send an audio stream to another process that synchronizes the audio playback with a video process. Delays in data transmission could cause audio and video to get out of sync.

5. Language Extensions

- Rather than creating entirely new programming languages, the authors of multimedia computing systems suggest **extensions** to existing languages to meet the needs of real-time processing. These extensions would support better handling of media types, files, and processes.
  - **Example:** Extensions could add real-time capabilities to languages like **C** or **ADA**, enabling the languages to handle continuous media more effectively. This may include special libraries or functions to manage multimedia streams, or features like parallel processing and inter-process communication.

Object-Oriented Approaches

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of “objects,” which can contain data and methods. It is useful in multimedia computing to model and manage complex data types such as audio, video, text, and graphics using object-based abstraction.

1. Class

- A **class** is a blueprint or template for creating objects.
- It is a **user-defined data type** and behaves like a built-in type.
- Contains:
  - **Data members**: variables

    - **Member functions**: methods that operate on data members

Syntax in C++

```
class class_name {
private:
   // data members and member functions
public:
   // data members and member functions
protected:
   // data members and member functions
};
```

2. Object

- An **object** is an instance of a class.
- Objects hold actual values and can perform functions defined in the class.
- Objects have:
    - **Identity**: unique name or reference
    - **State**: current values of its attributes
    - **Behaviour**: actions it performs (methods)

Example in C++

```
class person {
   char name[20]; // data members
   int id;
public:
   void getdata() { } // member function
};
int main() {
   person p1; // p1 is an object of class person
   return 0;
}
```

3. Inheritance

- **Inheritance** allows a class (child/derived class) to acquire properties and behaviors from another class (parent/base class).
- Promotes **code reusability**.
- Enables hierarchical classification.

Example

- Class: **Bird**
    - Subclass: **FlyingBird**
        - Sub-subclass: **Robin**
- A **Robin** “is-a” **Bird** — this forms an **is-a relationship**.

4. Polymorphism

- Polymorphism means **many forms**.
- It allows the same operation or method to behave differently on different classes.

Example

- **Addition Operation**:
    - 5 + 3 → 8 (integers)
    - "Hello" + "World" → "HelloWorld" (strings)
- Method Overriding:
    - findArea() in **Circle** and **Square** classes works differently depending on the shape.

Class Types in Multimedia Applications

In multimedia computing, **classes** are used to model various components like metaphors, devices, and processing units. These components are often represented as **objects** that encapsulate data and behaviors.

1. Application-Specific Metaphors as Classes

- These are **custom-designed classes** that represent **concepts unique to a specific application**.
- They help users interact using real-world analogies from the specific domain.

Example:

In a **music editing app**:

```
class Track {
   string instrument;
   float length;
   void play();
};
```

- Here, Track is a metaphor specific to that music app.

2. Application-Generic Metaphors as Classes

- These are **standard, reusable classes** used across multiple applications.
- They provide **common user-interface elements or structures**.

Example:

```
class Button {
   string label;
   void onClick();
};
```

- Button can be used in many multimedia apps (e.g., photo editor, video player, etc.).

3. Devices as Classes

- Devices like microphones, cameras, and display screens are modeled as classes.
- These classes abstract hardware behavior for interaction and control.

Example:

```
class Microphone {
   int volume;
   void startRecording();
   void stopRecording();
};
```

4. Processing Units as Classes

- These classes represent units that **process or transform multimedia data** like compression, filtering, rendering, etc.
- They encapsulate algorithms and logic.

Example:

```
class Compressor {
   int compressionRate;
   void compress(AudioFile file);
};
```

- Compressor handles multimedia compression independently.

5. Media as Classes

- In object-oriented multimedia systems, different **media types** (like text, image, audio, video) are modeled as **separate classes**.
- This allows **media-specific operations** to be encapsulated in their respective classes, promoting **modularity** and **reuse**.

Example:

```
class Audio {
  int duration;
  void play();
};
class Video {
  int resolution;
  void play();
};
```

- Both Audio and Video can have a play() method, but their internal workings differ (polymorphism).

6. Communication-Specific Metaphors as Classes

- Communication processes (like **sending**, **receiving**, **synchronization**, or **streaming**) are abstracted as classes.
- These classes help simulate real-world communication in software, making the system easier to design and understand.

Example:

```
class Stream {
  void sendData();
  void receiveData();
};
class Synchronizer {
  void syncAudioVideo();
};
```

- This helps in organizing **interactive** and **networked** multimedia applications.

Distribution of BMOs and CMOs

In object-oriented multimedia systems, **objects** can be categorized based on their role in multimedia processing and interaction. Two such types are:

1. BMO (Basic Multimedia Object)

- **Definition**: BMOs are the **lowest-level components** in a multimedia system.
- They represent **raw media elements**.

Examples:

- A video frame
- An audio sample
- A single image
- A text string

2. CMO (Composite Multimedia Object)

- **Definition**: CMOs are **higher-level objects** that are **composed of multiple BMOs** and sometimes other CMOs.
- They define **structured or synchronized multimedia presentations**.

Examples:

- A complete video scene with audio and subtitles
- A web page with embedded text, images, and animation
- A slideshow with narration

Distribution of BMOs and CMOs

In a **distributed multimedia system**, BMOs and CMOs may be **distributed across different systems or nodes** for performance, storage, or real-time processing.

Aspect	BMO	CMO
Granularity	Fine (raw media)	Coarse (structured content)
Distribution Level	Usually distributed near storage or hardware	Managed at application or presentation level

Data Volume	Small per object but high in number	Fewer objects, larger content
Synchronization	Requires real-time sync (e.g., video frames)	Requires structural sync (e.g., slides/audio)
Example (Distributed)	Audio samples streamed from a server	Full lesson combining slides, audio, and quiz

Why Distribution Matters

- **Efficient resource usage**: BMOs can be preloaded or streamed from different servers.
- **Scalability**: CMOs can reference BMOs from various sources.
- **Real-time performance**: Low-latency streaming and processing depend on optimized distribution.