Using Modern C++ to Build XOffsetDatastructure:

A Zero-Encoding and Zero-Decoding High-Performance Serialization Library in the Game Industry

FANCHEN SU

Cppcon
The C++ Conference

2024
September 15 - 20

About Me

- Fanchen Su
  - Game Research & Development Expert and Team Leader
  - Over 20 Years of Experience in C++ Programming
- Areas of Expertise:
  - Modern C++
  - Performance Optimization
  - Low Latency Systems
  - Code Maintainability
- Education:
  - Ph.D. in Computer Science, Wuhan University, 2012

Outline

- 1. Title
- 2. Why
- 3. What
- 4. Performance Statistics
- 5. How
- 6. Limitations and Plans
- 7. Summary and Takeaways

1. Title

The meaning of the title is explained in this section.

1. Title

- 1 Long Title:
  - Using Modern C++ to Build **XOffsetDatastructure**:
    A **Zero-Encoding and Zero-Decoding High-Performance** Serialization Library in the **Game Industry**
  - What
    - XOffsetDatastructure; Serialization Library; Game Industry.
  - Why
    - High Performance
  - How
    - Zero-Encoding & Zero-Decoding

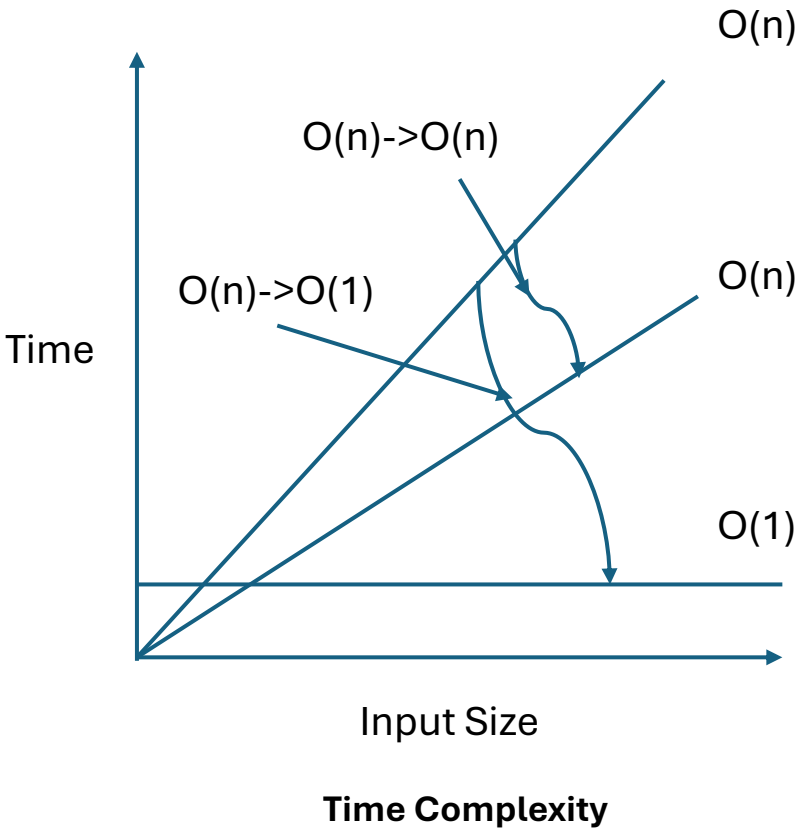
1. Title

- 1 Long Title:
  - Using Modern C++ to Build **XOffsetDatastructure**:
    A **Zero-Encoding and Zero-Decoding High-Performance** Serialization Library in the **Game Industry**
  - What
    - XOffsetDatastructure; Serialization Library; Game Industry.
  - Why
    - High Performance
  - How
    - Zero-Encoding & Zero-Decoding
- 3 Keywords:
  - XOffsetDatastructure, Zero-Encoding & Zero-Decoding and High-Performance.

1.1 XOffsetDatastructure

- **X** + Offset + Datastructure
- X
  - Extreme
- Design
  - O(n)->O(n) => O(n)->O(1)

// Instead of going from **O(n) to O(n)**, which offers limited benefits, we aim to transform from **O(n) to O(1)**, resulting in substantial efficiency gains.



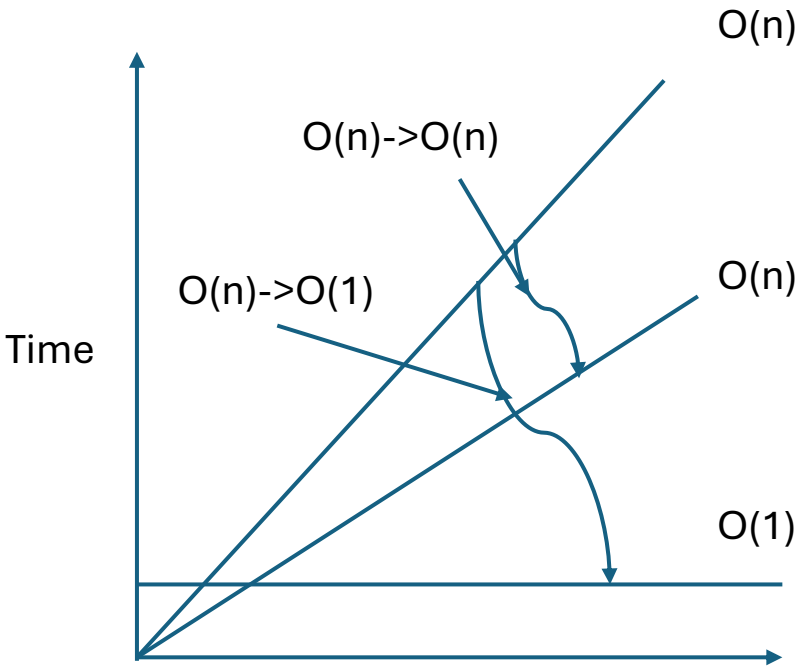
Time Complexity

1.1 XOffsetDatastructure

- **X** + Offset + Datastructure
- X
  - Extreme
- Design
  - O(n)->O(n) => O(n)->O(1)

  // Instead of going from **O(n) to O(n)**, which offers limited benefits, we aim to transform from **O(n) to O(1)**, resulting in substantial efficiency gains.
- Implementation
  - High performance

  // In terms of Implementation, we ensure **high performance** by optimizing how data is processed and managed.



Time Complexity

1.1 XOffsetDatastructure

- X + **Offset** + Datastructure
- Offset
  - **Offset-based Pointers**
  - **Memcopyable**

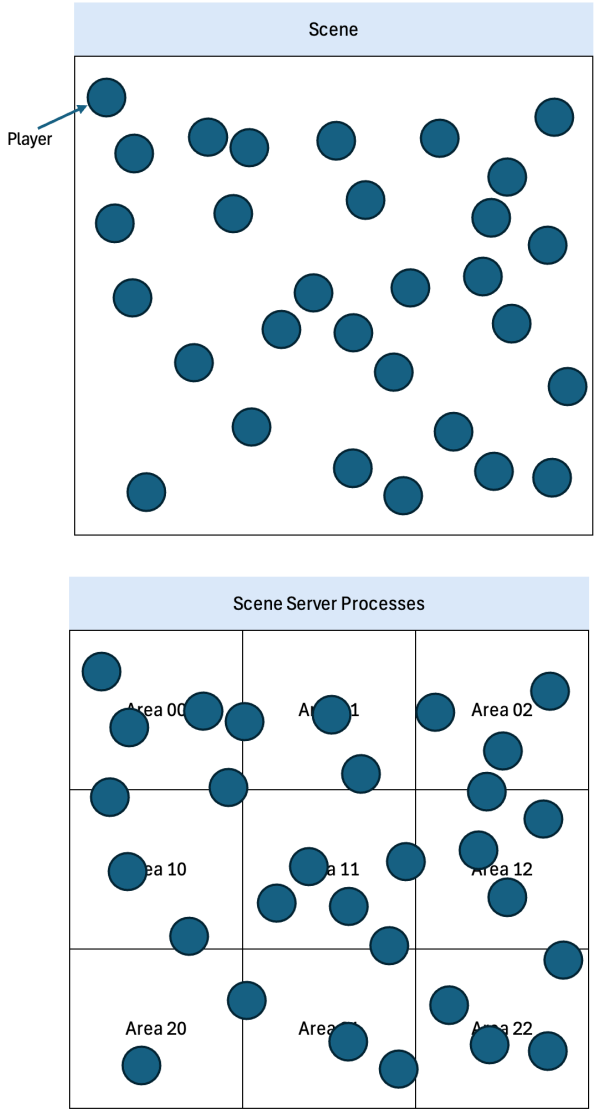
  // We use **Offset-based pointers** instead of **raw pointers**. This approach allows us to create data structures that are **memcopyable**, movable, and relocatable.

1.1 XOffsetDatastructure

- X + **Offset** + Datastructure
- Offset
  - **Offset-based Pointers**
  - **Memcopyable**

  // We use **Offset-based pointers** instead of **raw pointers**. This approach allows us to create data structures that are **memcopyable**, movable, and relocatable.
- Why do we need memcopyable structures?
  - Intra processes & inter processes
  - Intra devices & inter devices

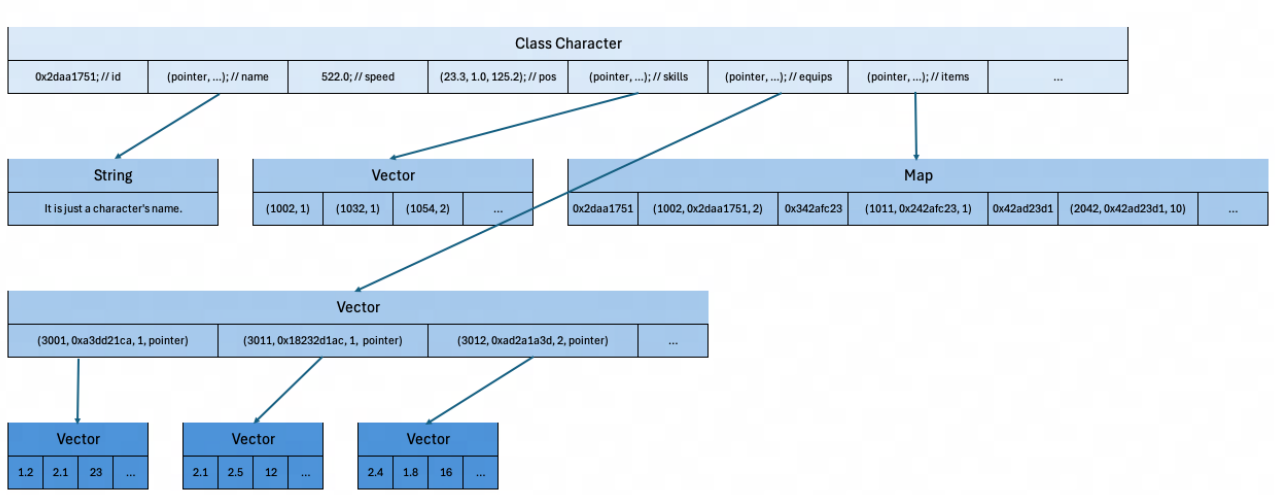
  // In large game environments with many players, a single process cannot handle everything. The game world is divided into smaller areas, each managed by its own process. As players move, data must migrate between these regions, leading to data transfers between **processes** and even **devices**.



Game Scene Division

1.1 XOffsetDatastructure

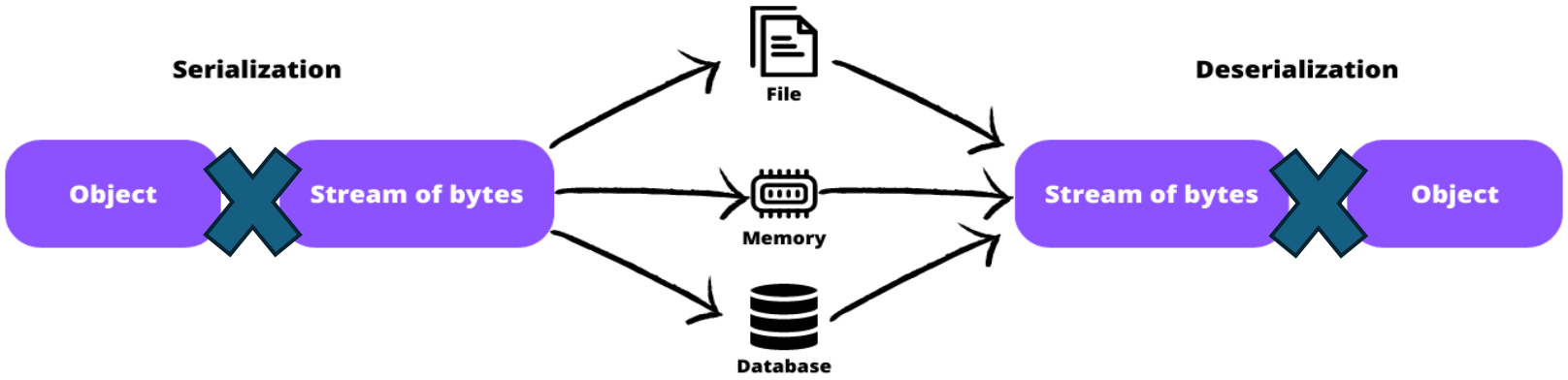
- X + Offset + **Datastructure**
- Data Structures
- XOffsetDatastructure is a collection of data structures, including:
  - **Base Types**, **Custom Types**, **Different Containers**, **Nested Types**.
- The diagram shows how these base types and containers can be combined to form **complex data structures**.



Complex Data Structure

1.2 Zero-Encoding & Zero-Decoding

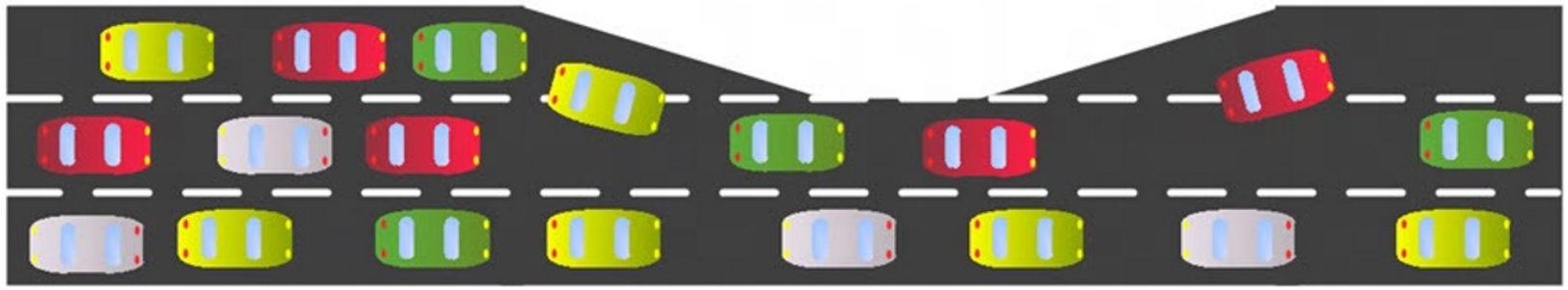
- **Serialization without encoding**: Data is stored and sent directly.
- **Deserialization without decoding**: Data is accessed and modified directly.
- The diagram shows that our method removes the need for encoding and decoding steps, simplifying the serialization and deserialization.



Zero-Encoding & Zero-Decoding

1.3 High-Performance

- In the game industry, **performance** is crucial and often the top priority. We need every operation to perform well to avoid any bottlenecks. These operations include:
  - **Serialization & deserialization**
  - **Read & write**
  - **In-place/non-in-place write**
- The diagram illustrates that, just like a road without bottlenecks allows for smooth traffic flow, we need to ensures that all operations run efficiently.



Read/Write Serialization Deserialization Read/Write

Performance Bottleneck

2. Why

The original intention of XOffsetDatastructure is presented in this section.

2.1 Serialization & Deserialization

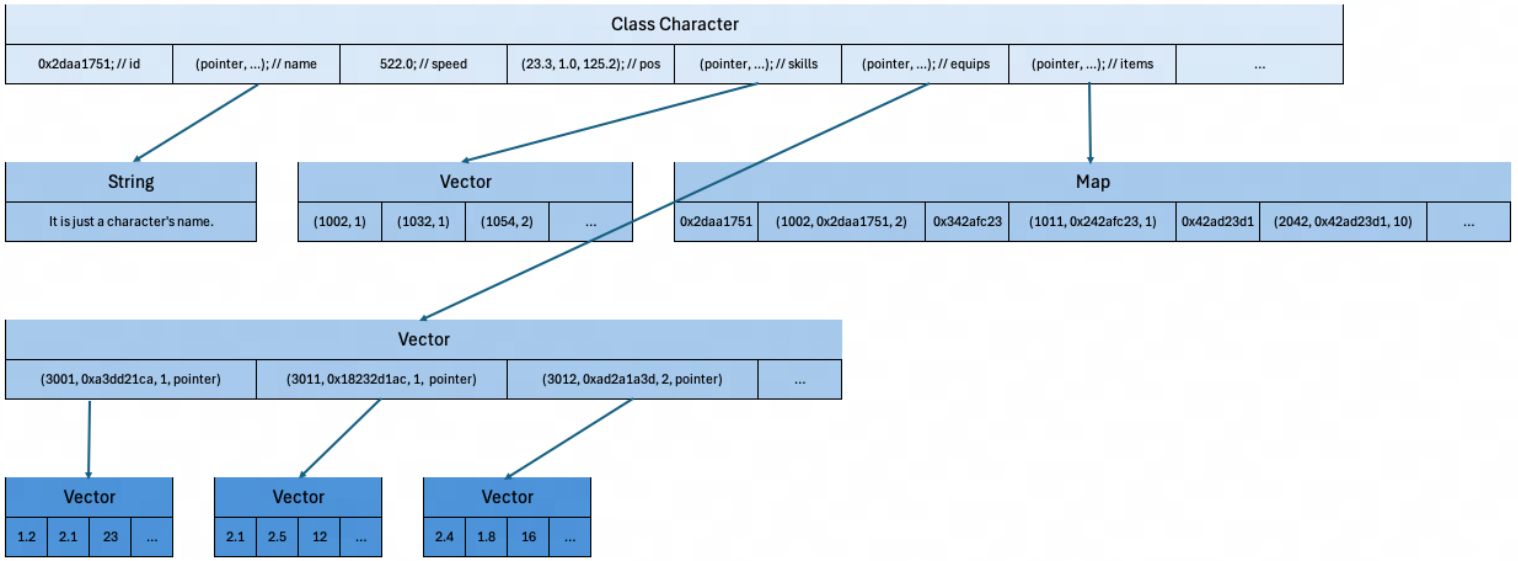
- What is serialization and deserialization?
  - In computing, serialization (or serialisation) is the **process** of translating a **data structure** or object state into a format that can be stored (e.g. files in secondary storage devices, **data buffers** in primary storage devices) or transmitted (e.g. data streams over computer networks) and reconstructed later (possibly in a different computer environment).
  - When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object.

2.1 Serialization & Deserialization

- What is serialization and deserialization?
  - In computing, serialization (or serialisation) is the **process** of translating a **data structure** or object state into a format that can be stored (e.g. files in secondary storage devices, **data buffers** in primary storage devices) or transmitted (e.g. data streams over computer networks) and reconstructed later (possibly in a different computer environment).
  - When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object.
- 3 Parts:
  - **Input**
  - **Process**
  - **Output**

2.1.1 What Is Serialization and Deserialization

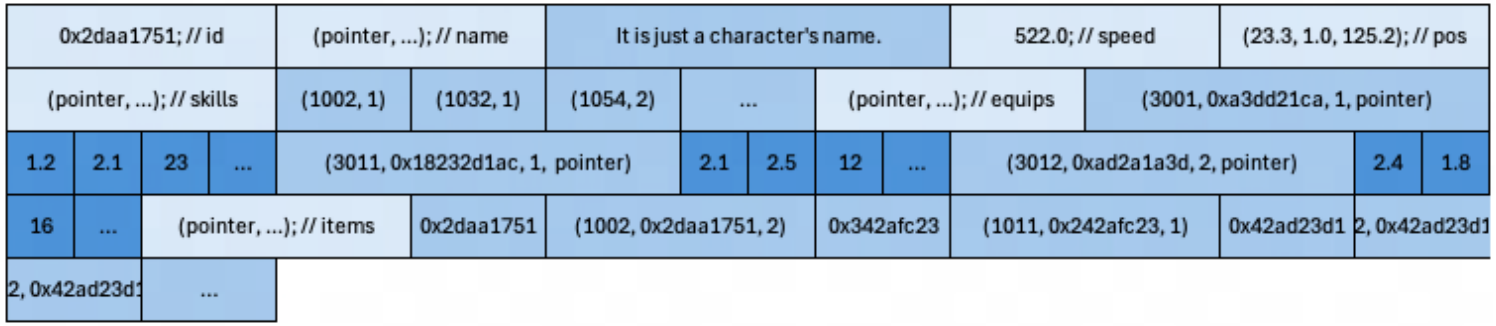
- Input
- The input for serialization is a **Data structure.**
- As shown in the diagram on the right, the lines connecting different elements represent **Pointers and References**, which is designed with **Time** and **Space** considerations.



Data Structure

2.1.1 What Is Serialization and Deserialization

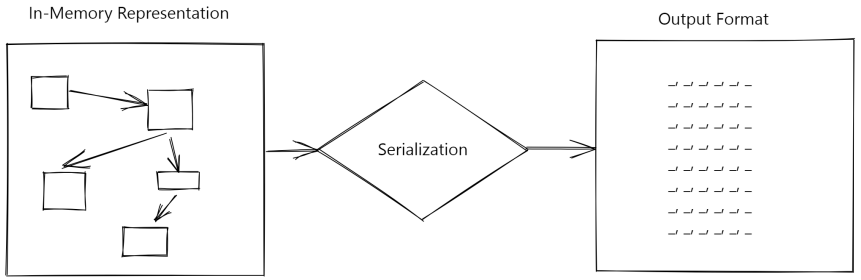
- Output
- The output is a **Data buffer**. The buffer is a **flat**, **contiguous** block of data designed for **storage** or **transmission**.
- As illustrated in the diagram, the data buffer is a single, uninterrupted space used to hold data.



Data Buffer

2.1.1 What Is Serialization and Deserialization

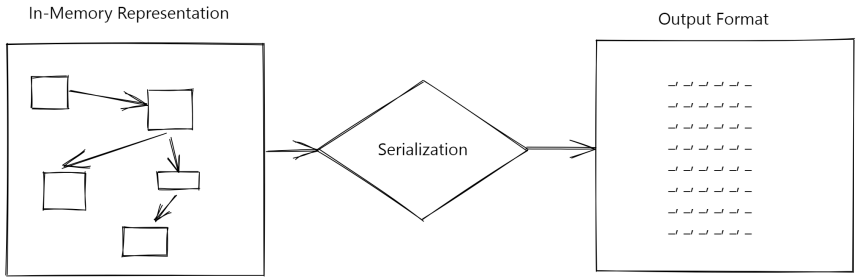
- Process
- The **Process** involves converting between structured data and flat data.
- **Serialization**:
  - This is the process of transforming structured data into flat data, as shown in the upper diagram.



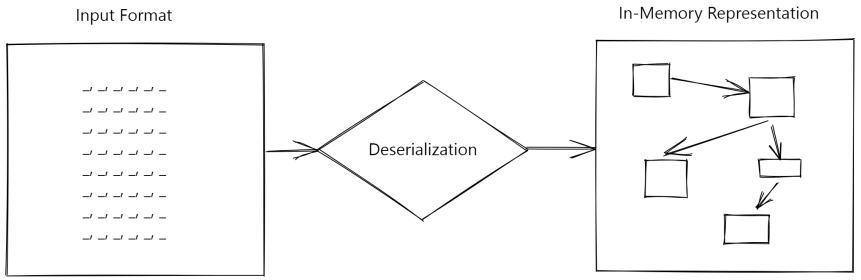
Serialization

2.1.1 What Is Serialization and Deserialization

- Process
- The **Process** involves converting between structured data and flat data.
- **Serialization**:
  - This is the process of transforming structured data into flat data, as shown in the upper diagram.
- **Deserialization**:
  - This is the reverse process, converting flat data back into structured data, as illustrated in the lower diagram.



Serialization



Deserialization

2.1.2 Why Is Serialization so Important

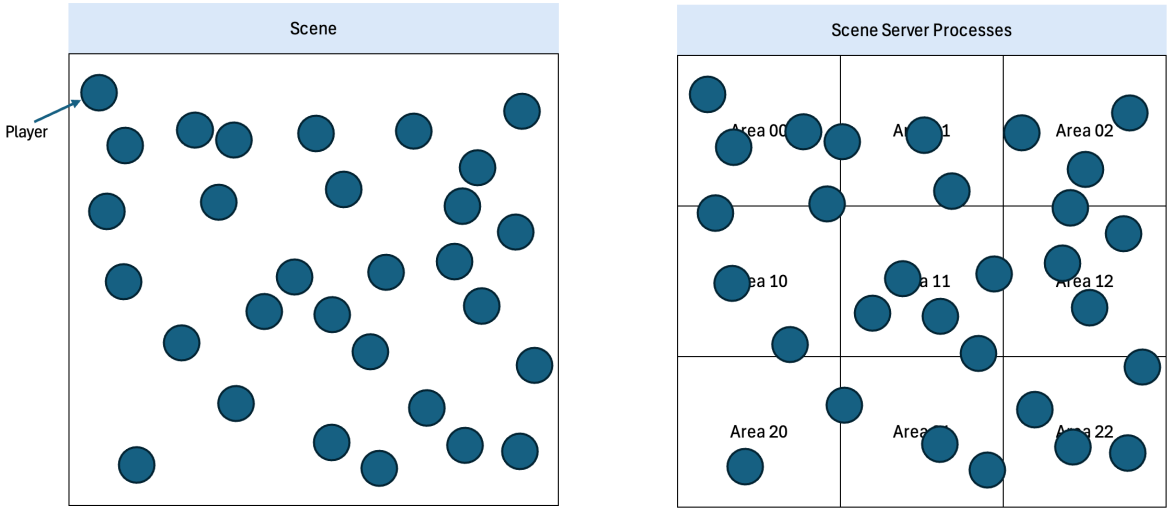
- Three key statistics highlight the importance of serialization:
- 12%
  - According to a study by Kanev et al., serialization and RPCs are responsible for **12%** of all fleet cycles across all applications at Google.
- 80-90%
  - According to another study by Palkar et al., modern big data applications can spend **80–90%** of CPU time parsing data.
- 20%
  - In the game industry, serialization/deserialization performance consumption has always been a factor that affects user experience and game performance.
  - In a typical commercial games, over **20%** of CPU time is spent on serialization and deserialization.

2.1.2 Why Is Serialization so Important

- Data communication in games occurs in several key areas:
- 1. Between servers and clients:
  - When entering a new scene or area, data for other players and NPCs must be loaded for display and interaction.
- 2. Server-to-server communication:
  - This includes RPC (Remote Procedure Calls) between different services.
- 3. Data migration:
  - Large amounts of game data need to be migrated among different scenes, areas, and lines.

2.1.2 Why Is Serialization so Important

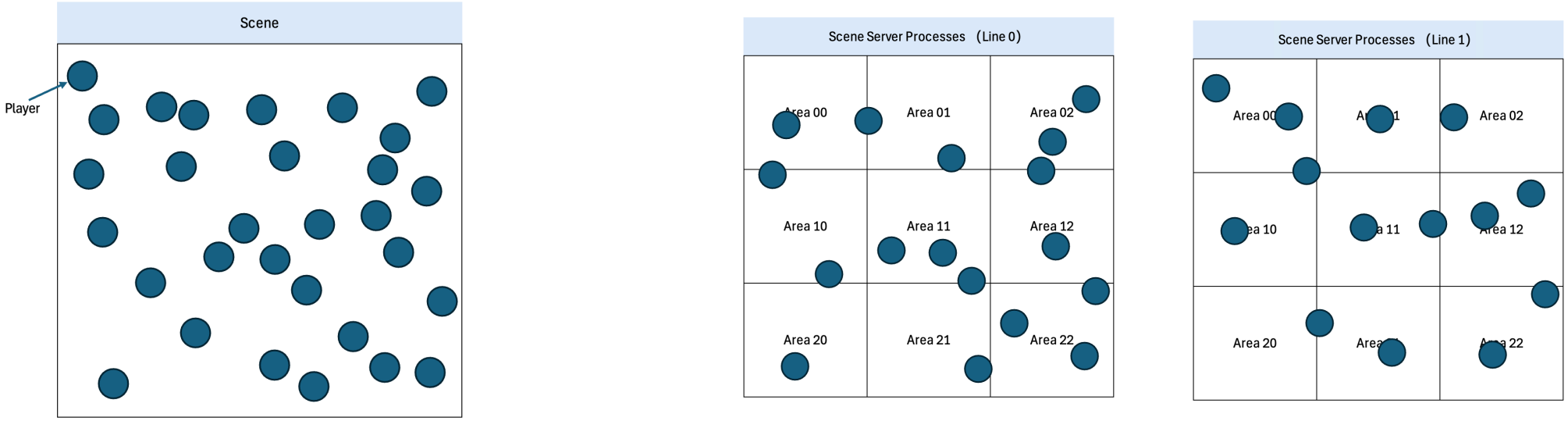
- Data Migration
- When a scene process becomes overloaded, it's split into areas and lines.
- As shown in the diagram, scenes are divided into different areas.
- Area: Spatial partition of a scene managed by one process.



Sence => Areas

2.1.2 Why Is Serialization so Important

- Data Migration
- When a scene process becomes overloaded, it's split into areas and lines.
- As shown in the diagram, players are distributed across different lines.
- A line is a logical grouping of players, managed by one process.



Scene => Lines

2.1.2 Why Is Serialization so Important

- Data migration
- Server Overloaded → Scene Division →Data Migration

// **Scene division** are crucial for managing **server load** and ensuring **smooth gameplay** even with **high player counts**.

// The division into areas and lines allows for more granular control over game data and processing, but it also increases the need for efficient data transfer between these divisions.

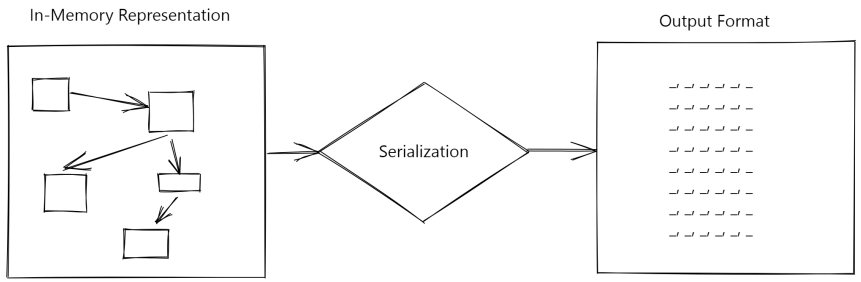
// As players move between areas or are redistributed across lines, their data needs to be quickly and accurately serialized, transferred, and deserialized.

// The contiguous **migration of game data** among scenes, areas, and lines requires a robust and efficient serialization system to maintain game state consistency and **performance**.

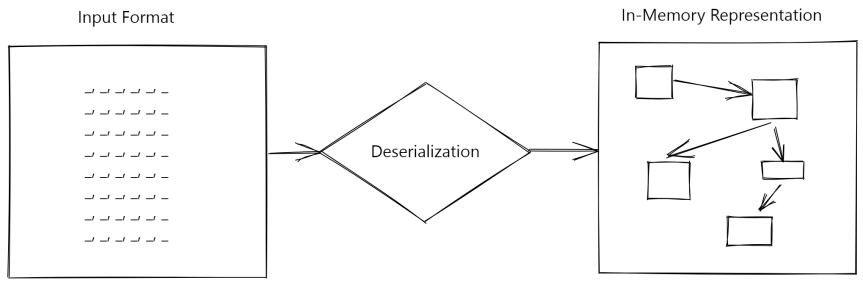
2.2 Current Solutions

- The 2 Categories
  - MessagePack, Protocol Buffers, etc.
  - FlatBuffers, Cap’n Proto, etc.
- **Input**, **Output**, and **Process**.

// To better understand these two categories of solutions, I'll briefly introduce them from three key aspects: **input**, **output**, and **process**.



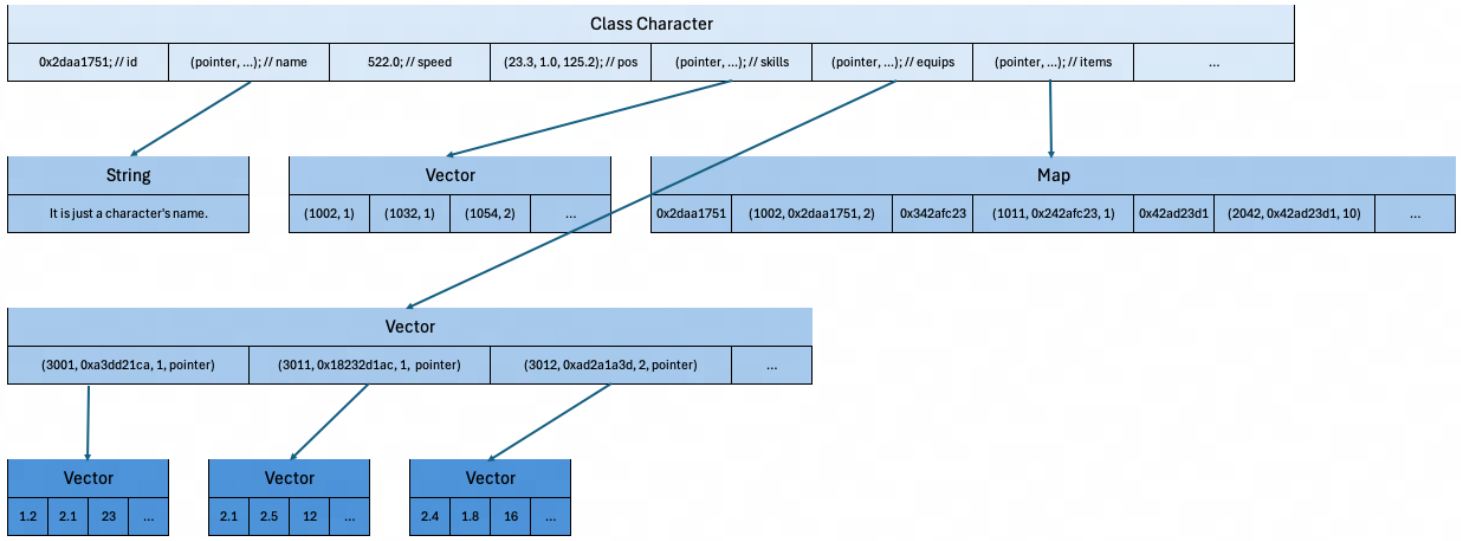
Serialization: Input, Process, Output



Deserilization: Input, Process, Output.

2.2.1 The First Category

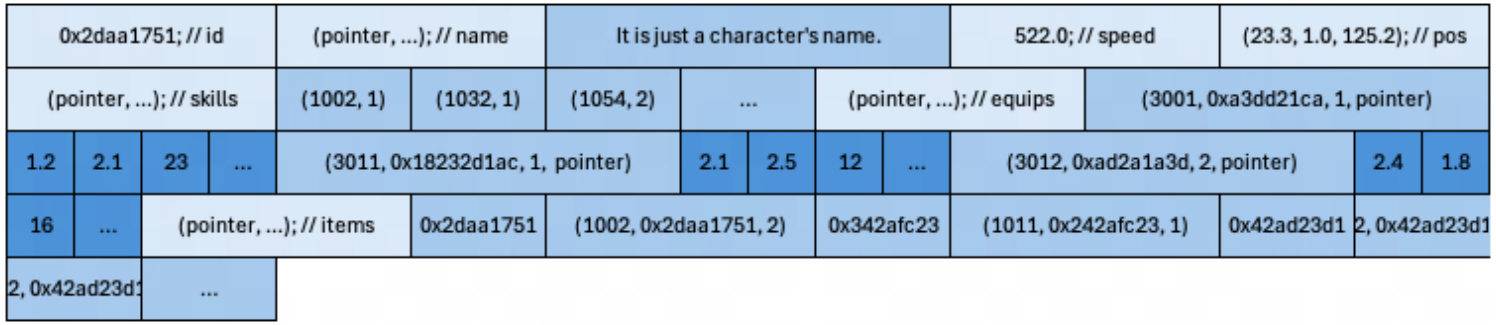
- Serialization
- The **input** in this case is the **memory data structure**.
- As shown in the diagram, the data in a memory data structure is distributed across various locations in memory, interconnected through pointers and references.



Memory Data Structure

2.2.1 The First Category

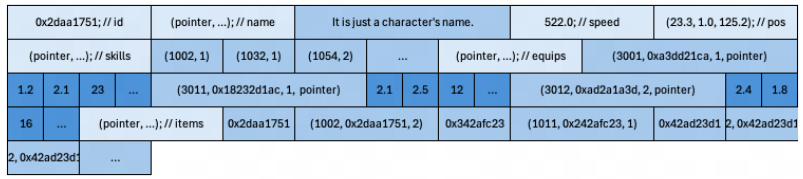
- Serialization
- The **output** is a **data buffer**.
- The **process** involves transforming the memory data structure into a data buffer. This transformation requires traversing and encoding all fields.
- Both **MessagePack** and **Protocol Buffers** follow this approach.



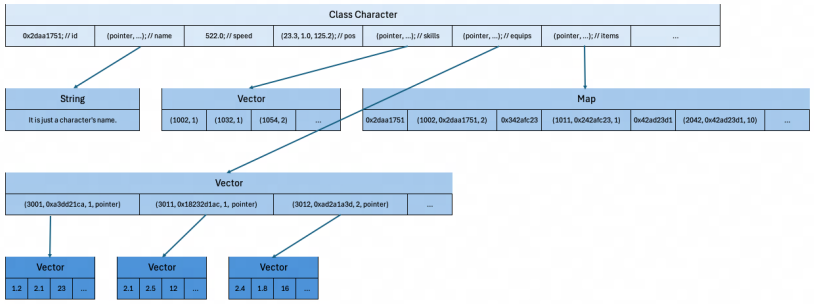
Data Buffer

2.2.1 The First Category

- Deserialization
- The **input** is the data buffer.
- The **output** is the memory data structure.
- The **process** entails converting the data buffer back into a memory data structure. This conversion needs traversing and decoding all fields.
- Both MessagePack and Protocol Buffers employ this method.



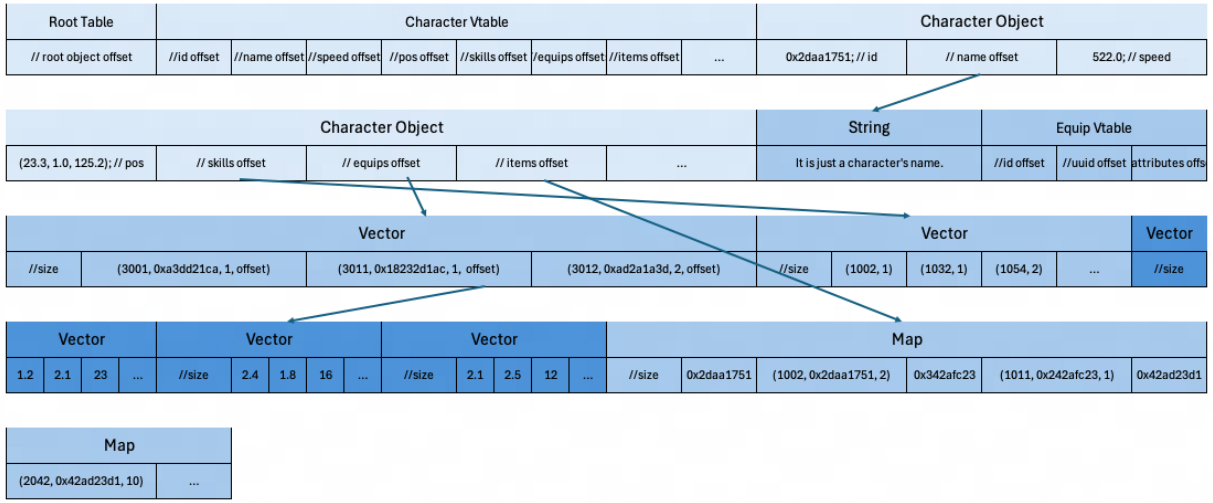
Data Buffer



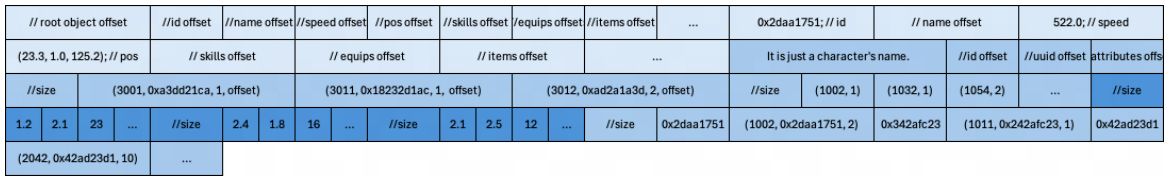
Memory Data Structure

2.2.2 The Second Category

- Serialization
- The **input** remains the same - it's the **memory data structure**.
- The **output** is the **serialization data structure**, which is similar to a data buffer.
- As shown in the diagram, the serialization data structure can convert to the data buffer directly and has a specific format for random access.



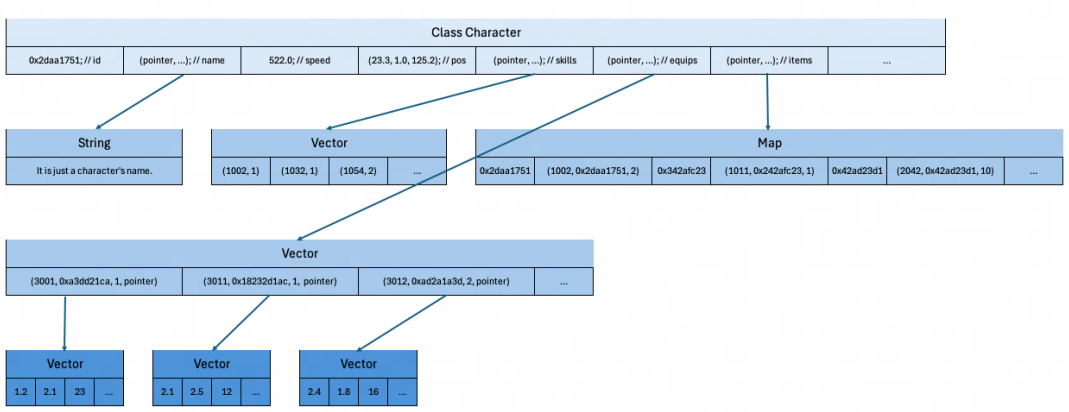
Serialization Data Structure



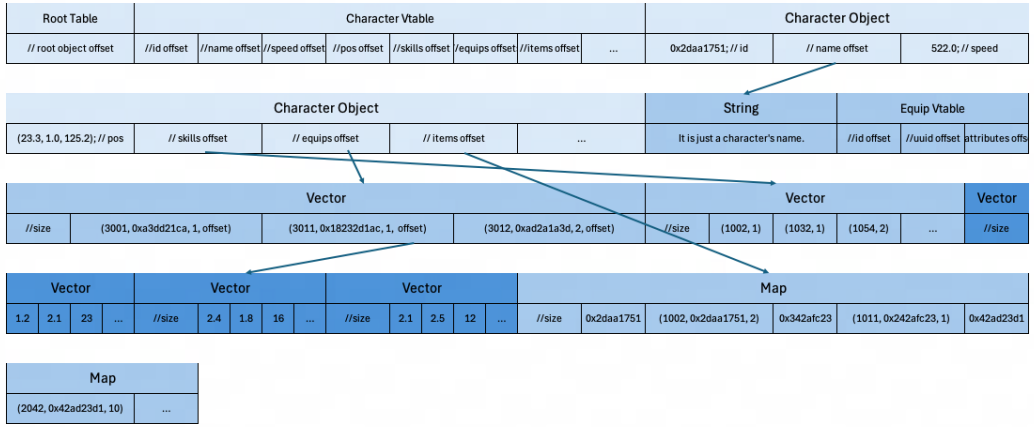
Data Buffer

2.2.2 The Second Category

- Serialization
- The **process** involves transforming the memory data structure into this serialization data structure, which becomes the data buffer.
- This transformation requires **traversing** and **setting** fields in memory using **Builders**.
- Examples of solutions that use this approach include FlatBuffers and Cap'n Proto.



Memory Data Structure



Serialization Data Structure

2.2.2 The Second Category

- Deserialization
- The **input** is the **data buffer**.
- The **output** is either the **serialization data structure** or the **memory data structure**.
- The **process** involves converting the data buffer into either the serialization data structure or the memory data structure.

2.2.2 The Second Category

- Deserialization
- The **input** is the **data buffer**.
- The **output** is either the **serialization data structure** or the **memory data structure**.
- The **process** involves converting the data buffer into either the serialization data structure or the memory data structure.
- In the **serialization data structure**, fields can be read directly. However, this structure **doesn't** allow for efficient in-place and non-in-place **modifications**.
- If efficient **modifications** are required, it becomes necessary to convert the structure field by field into a **memory data structure**.
- **FlatBuffers** and **Cap'n Proto** are examples of solutions that use this approach.

2.2.3 Current Solutions

- We've seen that serialization methods can be broadly divided into two categories:
  - The first category involves transforming a **memory data structure** into a **data buffer**.
  - The second category involves converting a **memory data structure** into a **serialization data structure** (**data buffer**).

2.2.3 Current Solutions

- We've seen that serialization methods can be broadly divided into two categories:
  - The first category involves transforming a **memory data structure** into a **data buffer**.
  - The second category involves converting a **memory data structure** into a **serialization data structure** (**data buffer**).
- Similarly, deserialization methods can also be categorized into two types:
  - The first type involves converting a **data buffer** back into a **memory data structure**.
  - The second type involves transforming a **data buffer** (**serialization data structure**) into a **memory data structure**.

2.2.3 Current Solutions

- We've seen that serialization methods can be broadly divided into two categories:
  - The first category involves transforming a **memory data structure** into a **data buffer**.
  - The second category involves converting a **memory data structure** into a **serialization data structure** (**data buffer**).
- Similarly, deserialization methods can also be categorized into two types:
  - The first type involves converting a **data buffer** back into a **memory data structure**.
  - The second type involves transforming a **data buffer** (**serialization data structure**) into a **memory data structure**.
- Despite their differences, these approaches share a common characteristic: they all require processing data **field by field**.
- This common feature leads us to a key concept that summarizes all these methods: **Traversal-Based Serialization**.
  - CppCon 16 Lightweight Object Persistence With Relocatable Heaps in Modern C++ by Bob Steagall

2.3 What & Why We Want to Do

- Our primary goal is to establish **equivalence** between **memory data structures** and **data buffers**. This equivalence would bring several significant advantages:
- For Serialization:
  - We want to remove the need for **encoding** fields one by one.
- For Deserialization:
  - We want to remove the necessity of **decoding** fields one by one.

2.3 What & Why We Want to Do

- Our primary goal is to establish **equivalence** between **memory data structures** and **data buffers**. This equivalence would bring several significant advantages:
- For Serialization:
  - We want to remove the need for **encoding** fields one by one.
- For Deserialization:
  - We want to remove the necessity of **decoding** fields one by one.
- Our goal is to enable **direct reading, writing** and **transmission**.
- Moreover, we‘re aiming for **high performance** across all operations:
  - serialization, deserialization, reading and writing, both in-place and non-in-place.

2.3 What & Why We Want to Do

- High-Performance and Easy to Use
  - Achieving these goals would make our solution suitable for use in **performance-sensitive** industries, such as the game industry. In such environments, the ability to **read and write freely**, along with **direct transmission** capabilities, is crucial.

2.3 What & Why We Want to Do

- High-Performance and Easy to Use
  - Achieving these goals would make our solution suitable for use in **performance-sensitive** industries, such as the game industry. In such environments, the ability to **read and write freely**, along with **direct transmission** capabilities, is crucial.
- Assumptions
  - **Trade-offs: Performance > Compatibility**.

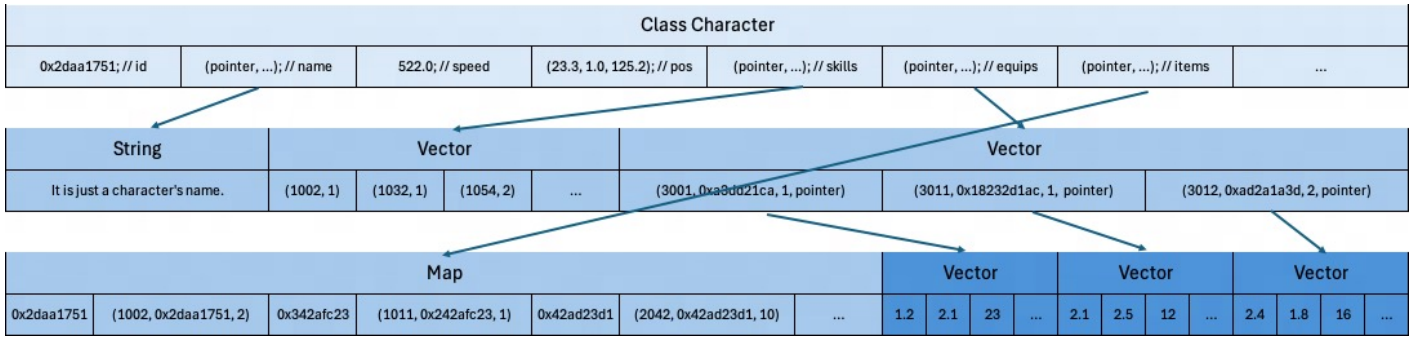
    // Now, it's important to note that we're working under certain assumptions and have made some **trade-offs**. Specifically, we've chosen to sacrifice some compatibility in favor of performance. This decision is based on the characteristics of certain industries.

    // For example, in the game industry, servers are typically homogeneous, which allows us to make certain assumptions about compatibility.

    // These design choices reflect our focus on maximizing performance for specific use cases, particularly in industries where the benefits of high performance outweigh the need for broad compatibility.
  - However, we're not entirely disregarding compatibility. We have some attempts to address compatibility issues, which we'll discuss later in the presentation.

2.4 Summary

- Serialization and Deserialization
  - Serialization and deserialization are extremely important processes in many industries.
- Current Solutions
  - These existing approaches don't fully meet the demands of high-performance environments like the game industry.
- What & Why We Want to Do
  - High-Performance
  - Free and direct reading/writing/transmission of data



XOffsetDatastructure

3. What

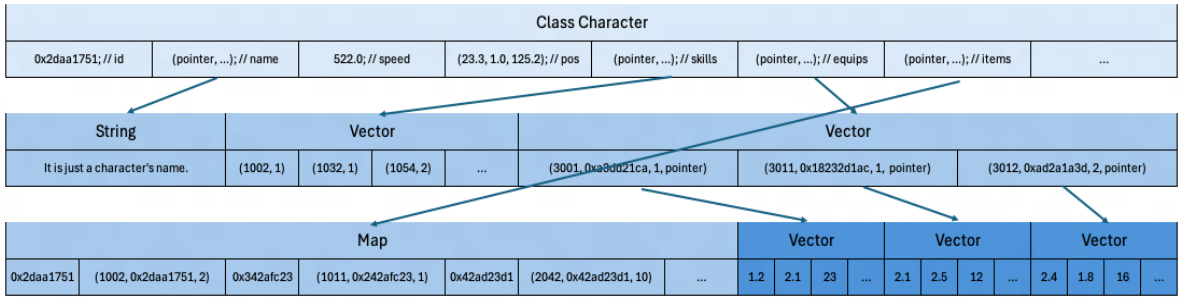
The main features of XOffsetDatastructure are presented in this section.

3.1 Zero-Encoding and Zero-Decoding

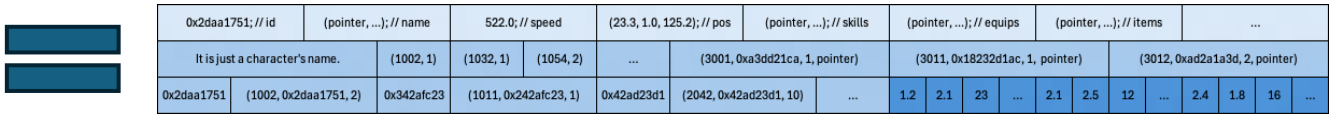
- Data can be Sent Directly
  - Data can be sent directly, **without** any additional **encoding** or **decoding** processes.

3.1 Zero-Encoding and Zero-Decoding

- Data can be Sent Directly
  - Data can be sent directly, **without** any additional **encoding** or **decoding** processes.
- Memory Data Structure == Data Buffer
  - This means that the **memory data structure** and the **data buffer** are equivalent.
- Contiguous Memory for Storage
  - As shown in the diagram, we achieve this by using **contiguous memory for storage**. This approach allows us to create a direct equivalence between the memory data structure and the data buffer.



Memory Data Structure

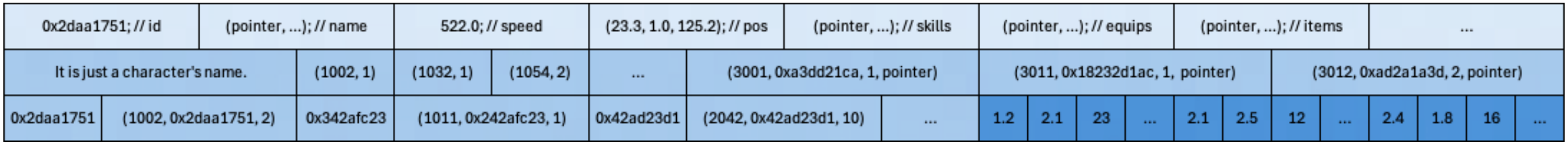


Data Buffer

3.1 Zero-Encoding and Zero-Decoding

- Zero-Encoding
- We can obtain data that‘s ready for **direct transmission** using the XTypeHolder::getBuffer method. This eliminates the need for any encoding step before sending the data.
- The code and diagram illustrate how this works in practice.

```
inline const std::vector<std::byte>& XTypeHolder::getBuffer() const
{
    return mBuffer; // std::vector<std::byte> mBuffer;
}
```

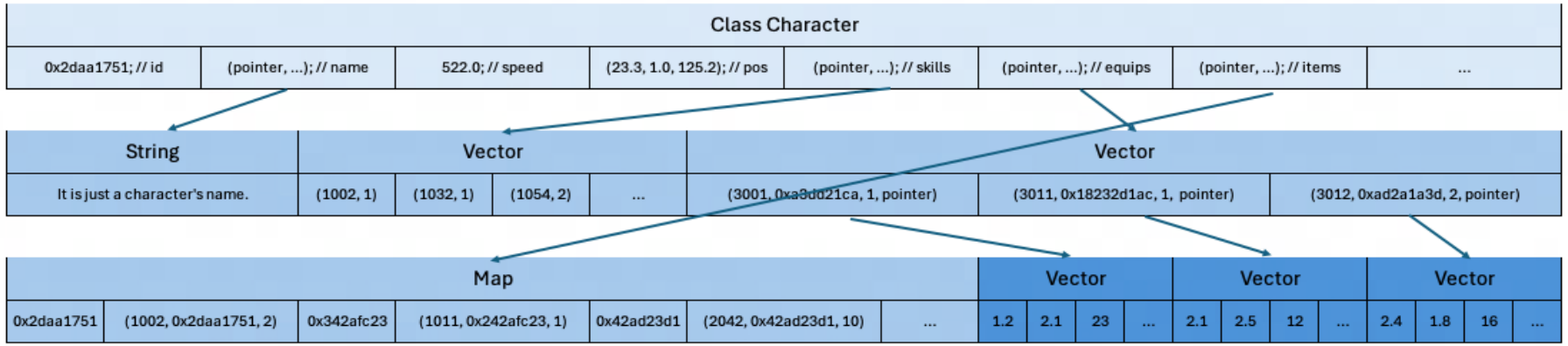


Data Buffer

3.1 Zero-Encoding and Zero-Decoding

- Zero-Decoding
- We can **read and write directly** to the data using the XTypeHolder::XTypeHolder method.
- The code and diagram demonstrate this process.

```
XTypeHolder::XTypeHolder(std::vector<std::byte> &externalBuffer)
    : mBuffer(std::move(externalBuffer)) // std::vector<std::byte> mBuffer;
{
    // ...
}
```



Memory Data Structure

3.2 Read and In-Place/Non-In-Place Write

- Read/[Non-]In-Place Write Directly
  - One of the key features of our approach is that fields can be **read** and **written directly**, including both **in-place** and **non-in-place** modifications.

```
XTypeHolder<Character> character;


character->speed = 522.0;
character->pos = {81.5, 12.3, 502.7};
character->skills.emplace_back(1054, 3);
character->items.emplace(std::piecewise_construct,
std::forward_as_tuple(0xda4ccb62c899d751),
std::forward_as_tuple(8056, 0xda4ccb62c899d751, 9
9, 0x171f7f8205fa4a1d));
character->items.erase(0xda4ccb62c899d751);
```

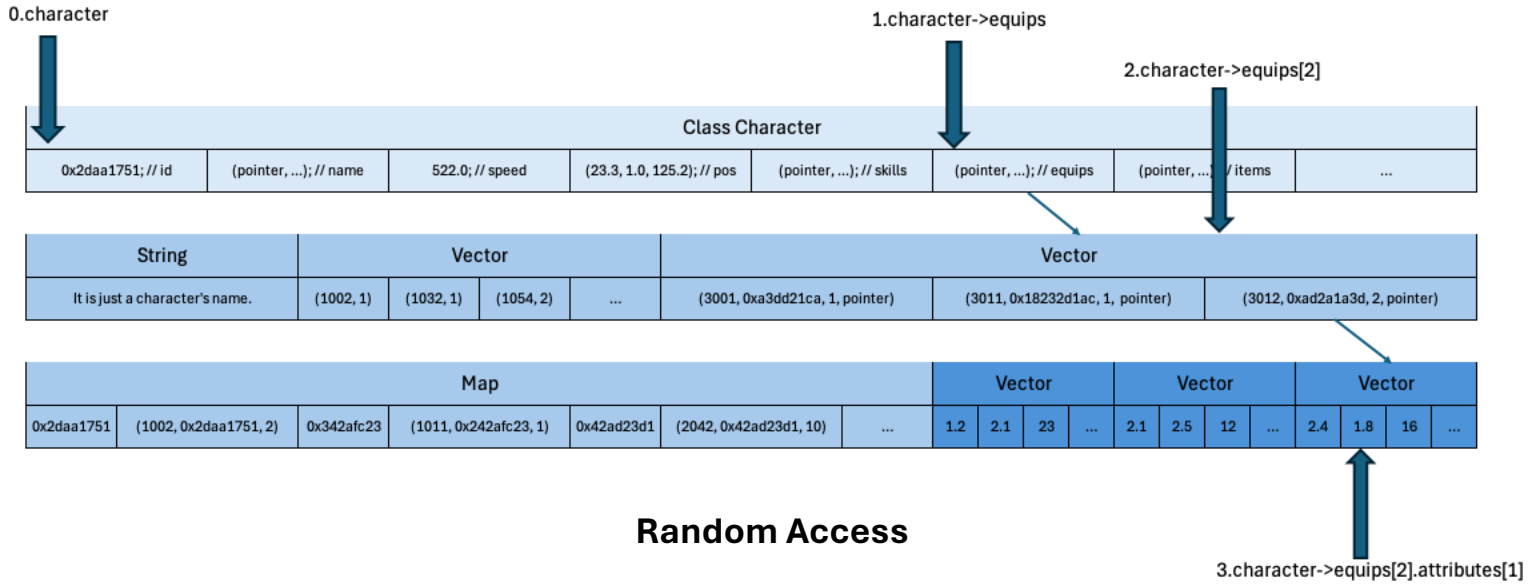
3.2 Read and In-Place/Non-In-Place Write

- Read/[Non-]In-Place Write Directly
  - One of the key features of our approach is that fields can be **read** and **written directly**, including both **in-place** and **non-in-place** modifications.
- Send Directly after Modifications
  - Importantly, all these operations maintain fields in contiguous memory, ensuring that the data can still be **sent directly** after multiple read and write operations.

```
XTypeHolder<Character> character;

character->speed = 522.0;
character->pos = {81.5, 12.3, 502.7};
character->skills.emplace_back(1054, 3);
character->items.emplace(std::piecewise_construct,
std::forward_as_tuple(0xda4ccb62c899d751),
std::forward_as_tuple(8056, 0xda4ccb62c899d751, 9
9, 0x171f7f8205fa4a1d));
character->items.erase(0xda4ccb62c899d751);

// get wire format with zero encoding
character.getBuffer();
```

3.2 Read and In-Place/Non-In-Place Write

- Read
- Our system allows for **random access** to data.
- As shown in the diagram, we can directly access nested data structures.
- For example, we can read ‘character->equips[2].attributes[1]‘ effectively.



3.2 Read and In-Place/Non-In-Place Write

- Write
- We support two types of write operations: **in-place** and **non-in-place**.
- In-Place Write:
  - This refers to modifying a data structure without allocating new memory or creating a new copy.
  - In-place writes are typically used for **fixed-size field modifications**.
- The code in bold in the diagram demonstrates an in-place write operation.

```
XTypeHolder<Character> character;

character->speed = 522.0;
character->pos = {81.5, 12.3, 502.7};
character->skills.emplace_back(1054, 3);
character->items.emplace(std::piecewise_construct,
std::forward_as_tuple(0xda4ccb62c899d751),
std::forward_as_tuple(8056, 0xda4ccb62c899d751, 9
9, 0x171f7f8205fa4a1d));
character->items.erase(0xda4ccb62c899d751);

// get wire format with zero encoding
character.getBuffer();
```

3.2 Read and In-Place/Non-In-Place Write

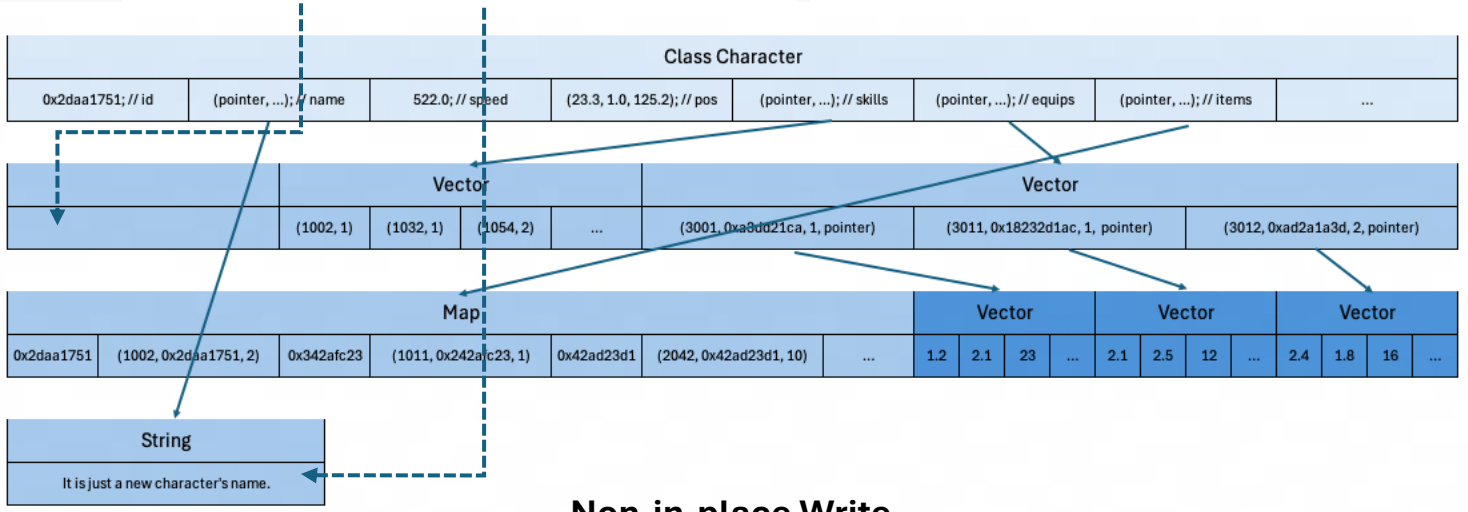
- Write
- Non-In-Place Write:
  - This type of write operation is generally used for **variable-size field modifications**.
- The code in bold in the diagram shows examples of non-in-place write operations.
  - Vector
  - Map
  - String
  - …

```
XTypeHolder<Character> character;

character->speed = 522.0;
character->pos = {81.5, 12.3, 502.7};
character->skills.emplace_back(1054, 3);
character->items.emplace(std::piecewise_construct,
std::forward_as_tuple(0xda4ccb62c899d751),
std::forward_as_tuple(8056, 0xda4ccb62c899d751, 9
9, 0x171f7f8205fa4a1d));
character->items.erase(0xda4ccb62c899d751);

// get wire format with zero encoding
character.getBuffer();
```

3.2 Read and in-place/non-in-place Write

- The **challenge** with non-in-place writes is maintaining a contiguous memory layout after the modifications.
- Let's look at an example where we modify the character's name twice. The corresponding memory layout changes are illustrated in the diagram.

```
character->name = "It is just a character's name.";
character->name = "It is just a new character's name.";
```

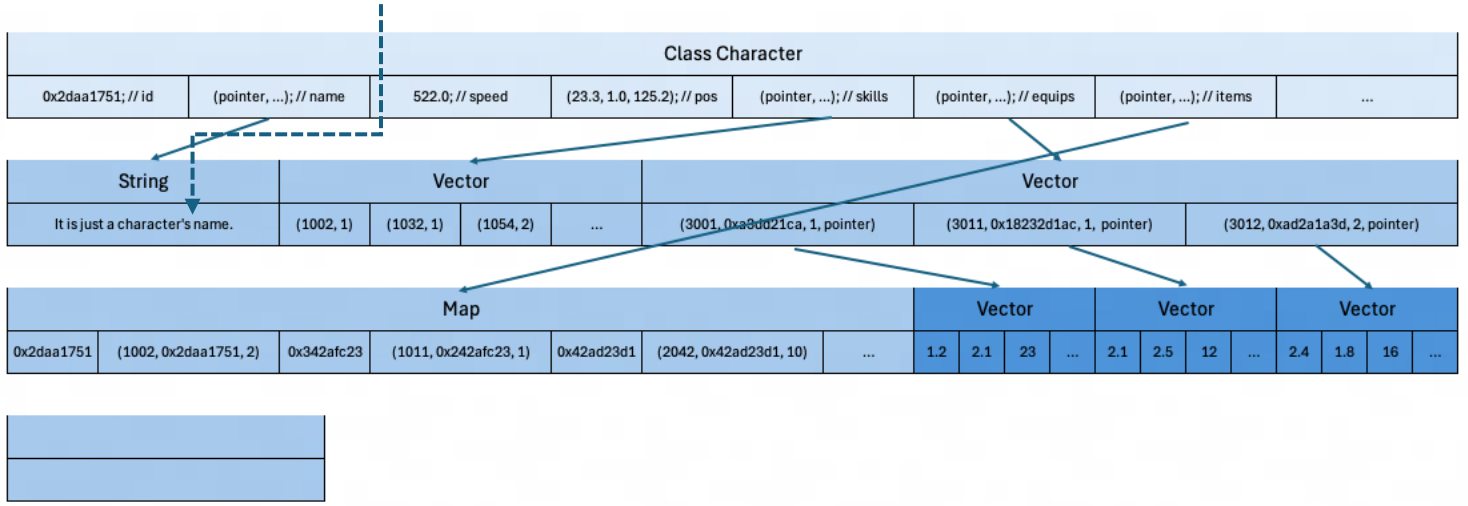


Non-in-place Write

3.2 Read and In-Place/Non-In-Place Write

- The **challenge** with non-in-place writes is maintaining a contiguous memory layout after the modifications.
- Let's look at an example where we modify the character's name twice. The corresponding memory layout changes are illustrated in the diagram.

```
...;
character->name = "It is just a character's name.";
```



Non-in-place Write

3.3 Various Types and Containers

- All base types, custom types, various containers, and nested types are supported.
- **Base Types**:

  These include fundamental data types such as int, float, and so on.

  ```
  uint64_t id;
  ```

- **Custom Types**:

  We support user-defined structures and classes, allowing for complex data representations.

  ```
  struct Item
  struct Equip : public Item
  ```

- **Containers**:

  Our system can handle container types, such as vector of float values.

  ```
  XVector<float> attributes;
  ```

3.3 Various Types and Containers

- All base types, custom types, various containers, and nested types are supported.
- **Nested Types**:

  We support nested combinations of base types, custom types, and containers, enabling complex data structures.

```
XVector<Equip> equips;
XMap<uint64_t, Item> items; Code sample
struct Character
{
    // ...
    XMap<uint64_t, Item> items;
};
```

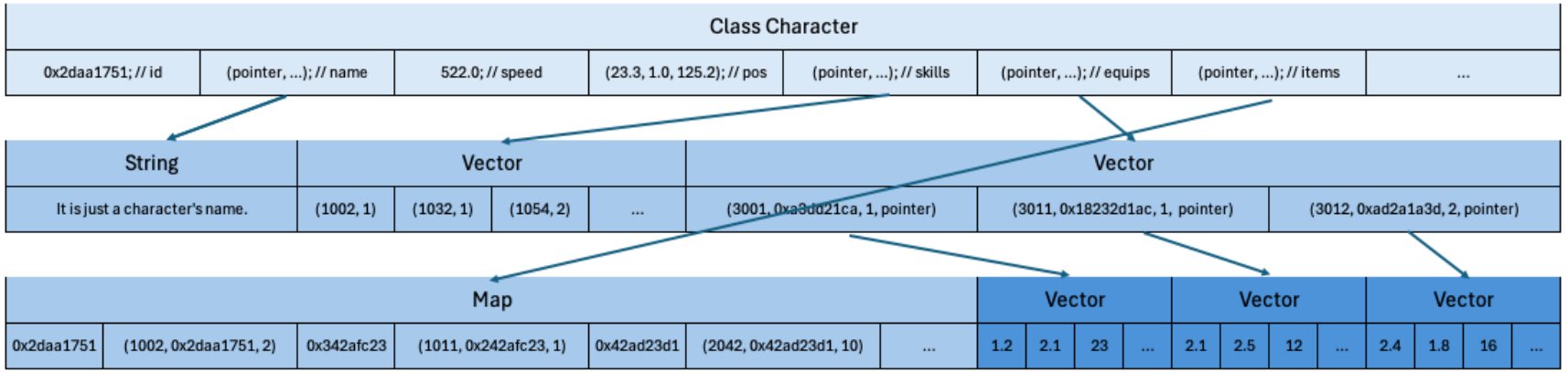
3.3 Various Types and Containers

- A Practical Example
- As shown in the diagram, we have a ‘Character' type. This character data structure incorporates:
  - Base types (like integers for id)
  - Containers (such as vectors for inventory items)
  - Custom types (like equipment structures)
  - Nested types (combinations of the above, such as a vector of custom equipment types)

```
struct Item
{
    uint32_t id;
    // ...
};
struct Equip : public Item
{
    uint32_t level;
    XVector<float> attributes;
};
struct Character
{
    uint64_t id;
    XString name;
    // ...
    Position pos;
    XVector<float> attributes;
    XVector<Equip> equips;
    XMap<uint64_t, Item> items;
};
```

3.3 Various Types and Containers

- Memory Layout
- The following diagram shows how this complex ‘Character' structure is laid out in memory.
- Despite its complexity, you can see that all the data is stored in a **contiguous memory block**. This contiguous layout is crucial for our zero-encoding/decoding approach and enables efficient data transmission.



XOffsetDataStructure

3.4 High Performance Read and Write

- Read/Write Performance Comparison
- Our benchmarks show that its **performance** is **comparable** to, and in some cases even slightly exceed, that of STL and Boost Container libraries.
- As you can see in the chart, our solution demonstrates a slight performance advantage in read and write operations. This performance is primarily attributed to our use of a custom allocator.

Containers	Read/Write(100 thousand times, ms)
std::*	68.0446
boost::*	56.273
XOffsetDatastructure::*	49.8465

Read/Write Performance

3.4 High Performance Read and Write

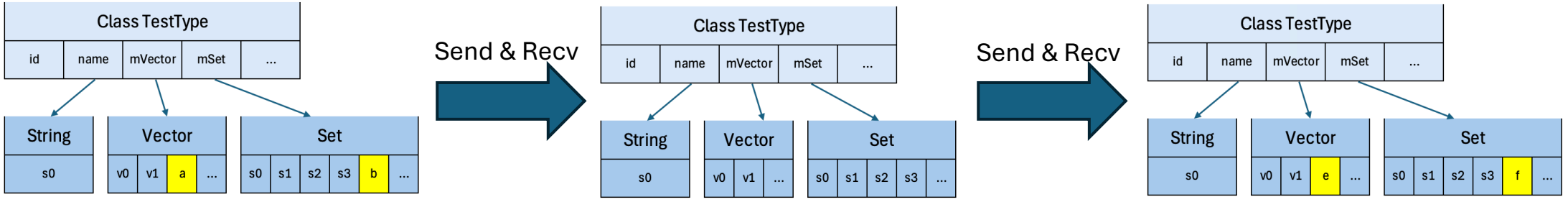
- Use as Base Data Structures in Memory
  - This high level of performance has significant implications for practical applications. Due to its excellent read and write speeds, XOffsetDatastructure can be used directly as base **data structures in memory**.
- The ability to use these high-performance structures directly in memory, without sacrificing serialization capabilities, is particularly valuable in performance-critical applications.
- It allows for rapid data access and modification while still maintaining the ability to quickly serialize and transmit data when needed.

3.5 Summary

- Our solution offers several key advantages:
  - Complex Data Structure Representation
  - High-Performance Read and Write Operations
  - Zero Encoding and Zero Decoding

3.5 Summary

- Our solution offers several key advantages:
  - Complex Data Structure Representation
  - High-Performance Read and Write Operations
  - Zero Encoding and Zero Decoding
- As illustrated in the diagram, our solution enables free and efficient flow of data through all stages: reading, writing, sending and receiving data.
- The ability to freely and efficiently read, write, send, and receive data in this manner opens new possibilities for how we can design and implement high-performance systems, particularly those dealing with complex, rapidly changing data structures, such as in real-time game applications.



Data Flow

4. Performance Statistics

The performance statistics of XOffsetDatastructure are presented in this section.

4.1 Test Cases

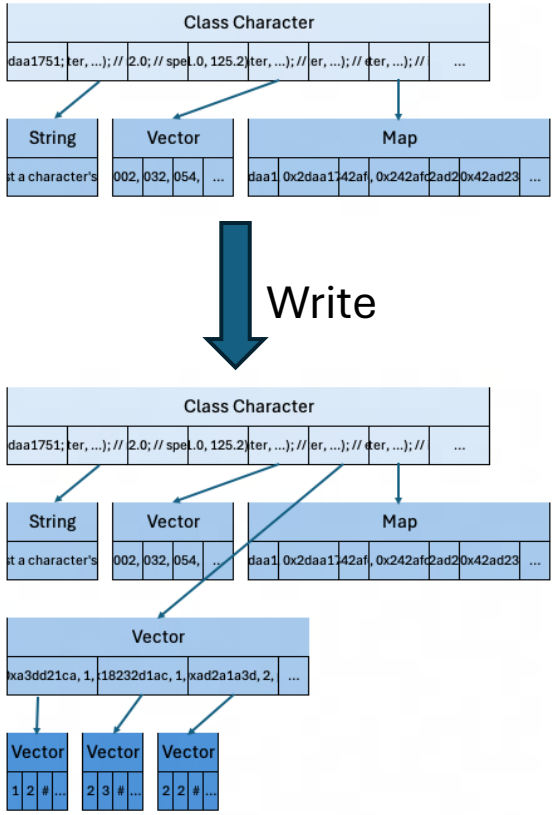
- Real-World Data Type
  - For our performance evaluations, we've chosen to use a complex, real-world data type that's common in game applications: class **Character**.
- Entire Lifecycle
  - We test the performance of various operations throughout the **entire lifecycle** of the data. Our tests cover a wide range of operations, including: Reading, Writing, Serialization and Deserialization.

4.1 Test Cases

- Real-World Data Type
  - For our performance evaluations, we've chosen to use a complex, real-world data type that's common in game applications: class **Character**.
- Entire Lifecycle
  - We test the performance of various operations throughout the **entire lifecycle** of the data. Our tests cover a wide range of operations, including: Reading, Writing, Serialization and Deserialization.
- To ensure the validity and consistency of our comparisons, we've used the **same random sequence** across all test cases.
- Compiler: MSVC 1937.
- Cpu & Mem: Intel i9-13900K 24 cores &128G.

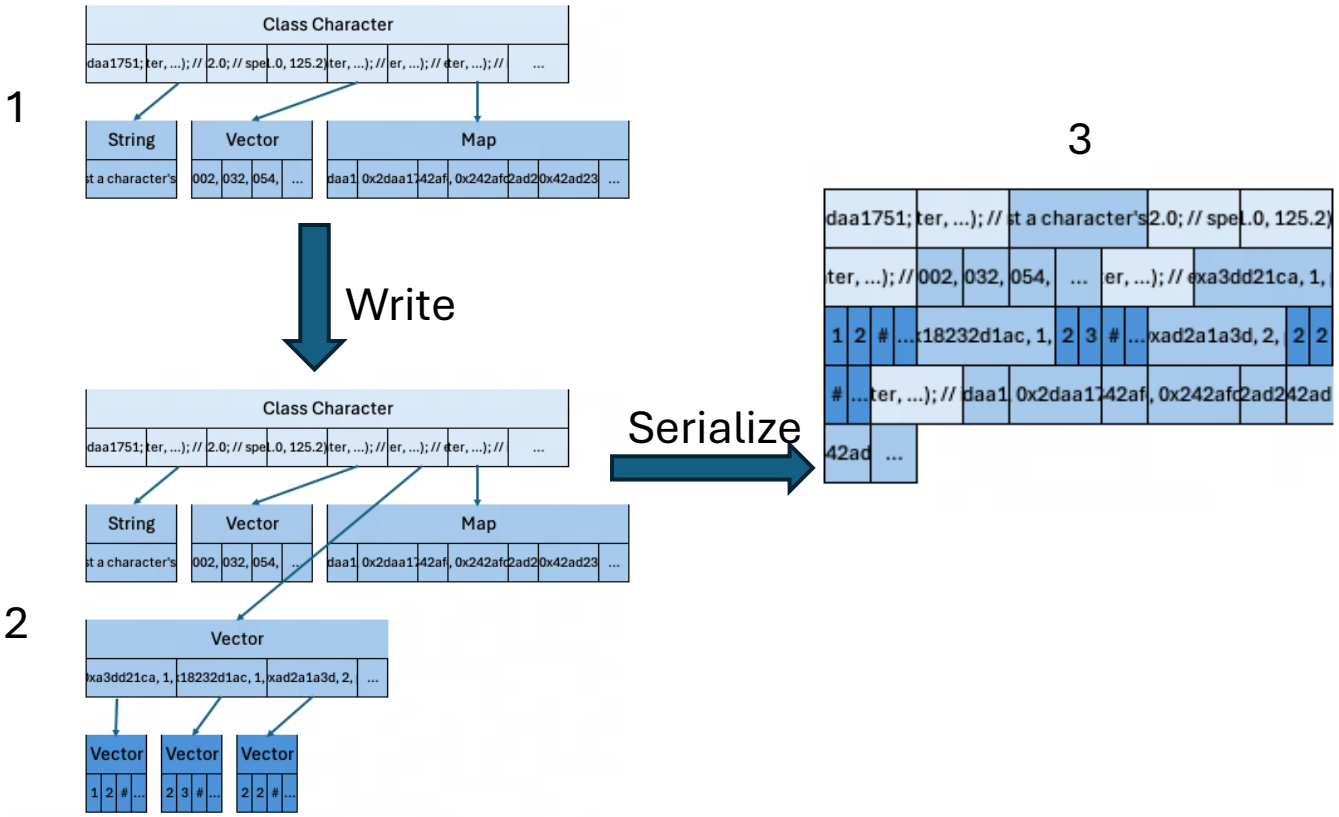
4.1 Test Cases

1



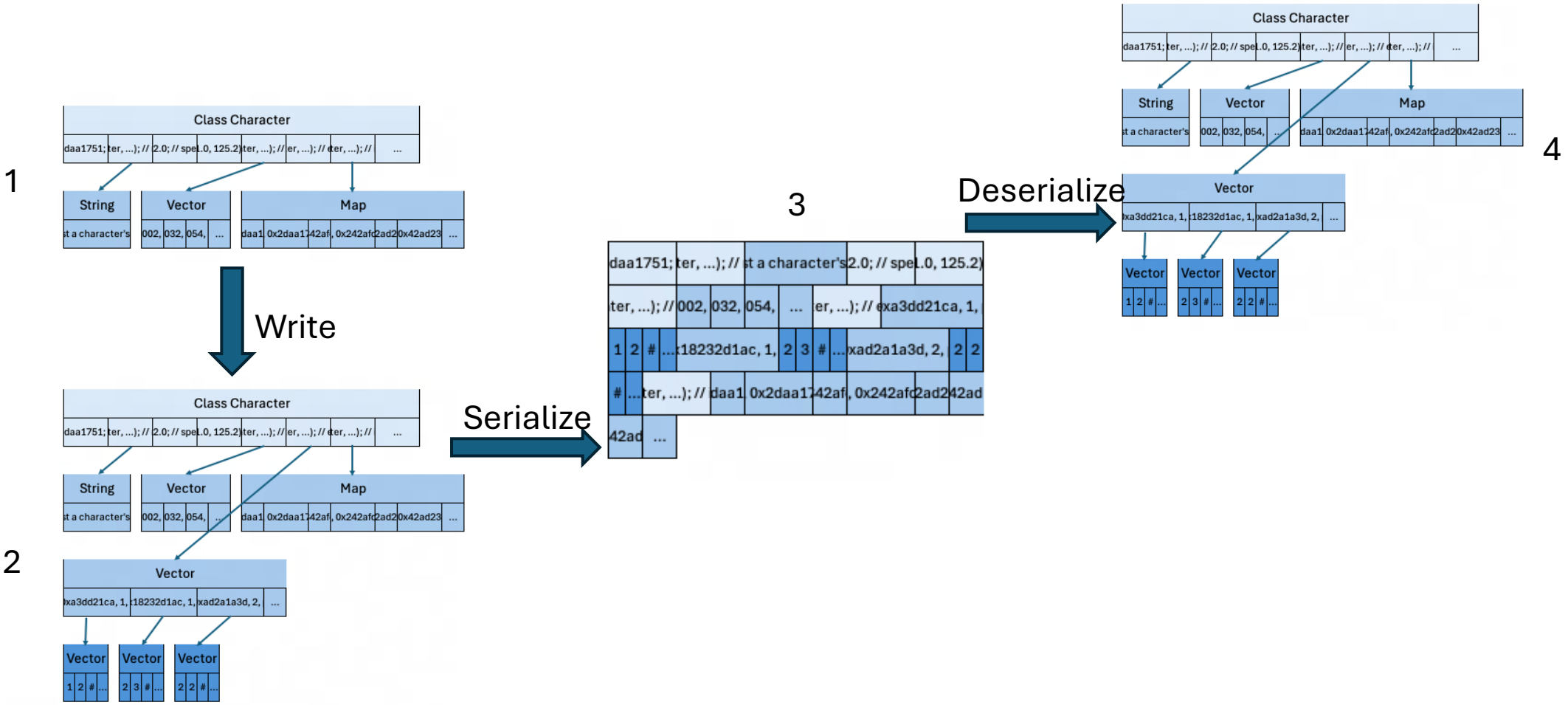
Data Flow Across Processes/Devices

4.1 Test Cases



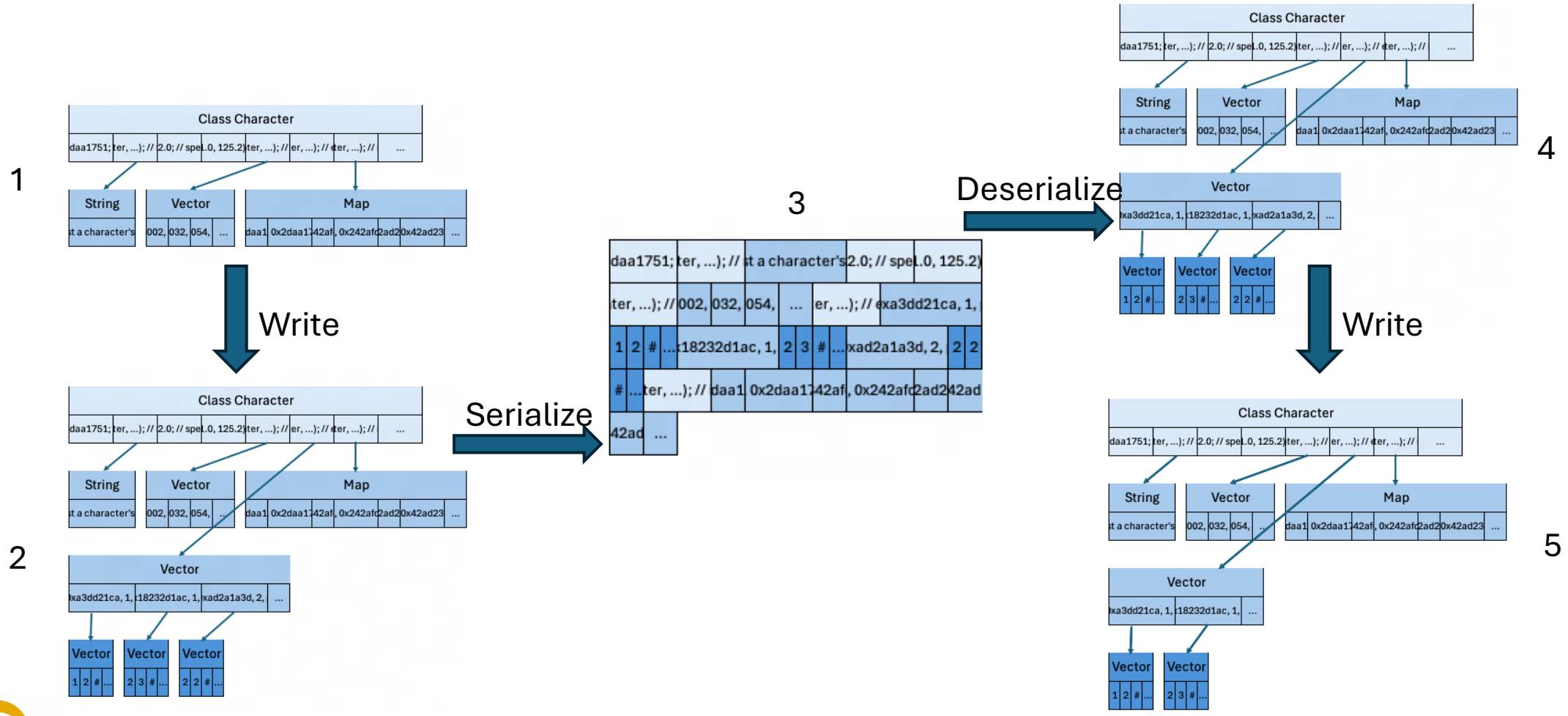
Data Flow Across Processes/Devices

4.1 Test Cases



Data Flow Across Processes/Devices

4.1 Test Cases



Data Flow Across Processes/Devices

4.2 Encoding and Decoding

- The advantage of XOffsetDatastructure in terms of encoding and decoding performance is evident in our tests.
- Due to our zero-encoding and zero-decoding approach, the time required for these operations is almost zero. This represents a significant performance improvement over traditional serialization methods.

Algorithms	Encoding/Decoding(100 thousand times, ms)
Cap'n Proto	259.225
FlatBuffers	679.442
MessagePack	957.7
XOffsetDatastructure	**0.0530667**
Protocol Buffers	1578.06

Encoding/Decoding Performance

4.3 Read and Write Performance

- XOffsetDatastructure has advantages in read and write performance.
- The most important implication of this high performance is that XOffsetDatastructure can be used directly as the base **memory data structure** in applications.

Algorithms	Containers Implementation	Read/Write (100 thousand times, ms)
Cap'n Proto	std::*	68.0446
FlatBuffers	std::*	68.0446
MessagePack	std::*	68.0446
XOffsetDatastructure	**XOffsetDatastructure::***	**49.8465**
Protocol Buffers	google::protobuf::*	56.5378

Read/Write Performance

4.4 Message Size

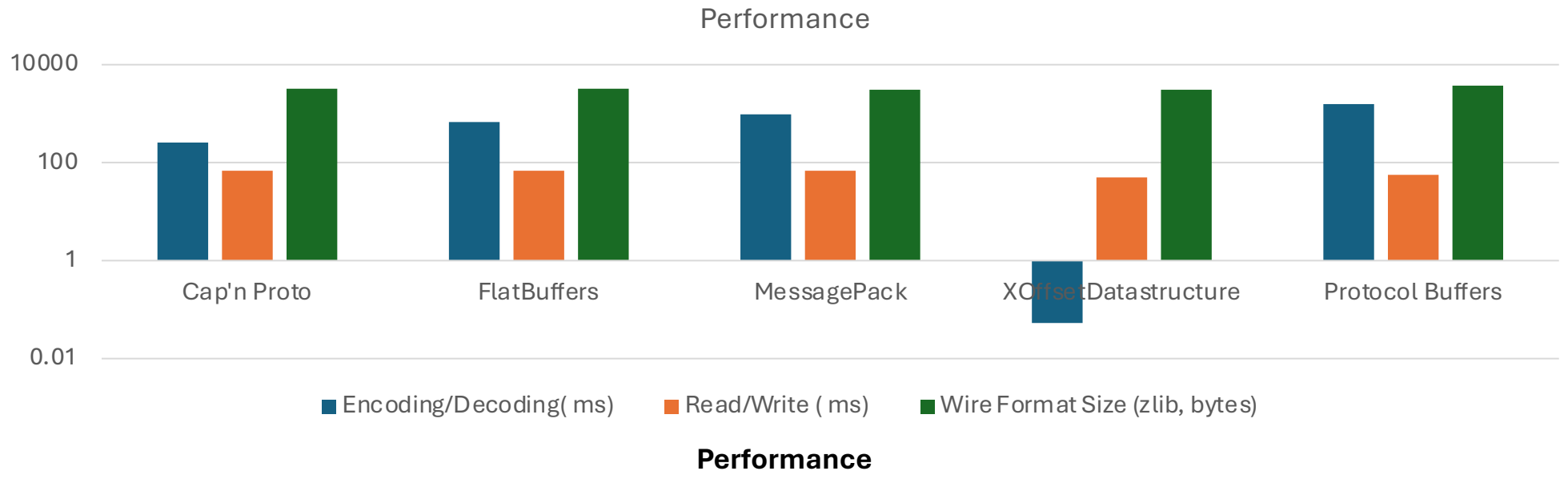
- The message size comparison is shown in the table below. The difference between the compressed message sizes is not significant.
- The compressed message size is crucial because network messages are typically transmitted in compressed form.

Algorithms	Wire Format Size (zlib/normal, bytes)
Cap'n Proto	3205/3656
FlatBuffers	3247/3672
MessagePack	3064/3222
XOffsetDatastructure	**3075/4184**
Protocol Buffers	3759/4100

Message Size

4.5 Summary

- These results collectively demonstrate the potential advantages of our XOffsetDatastructure approach: Near-zero encoding and decoding times, Better read and write performance and comparable message sizes.
- These characteristics suggest that our solution could be well-suited for applications requiring efficient data handling and transmission, such as in the game industry.



5. How

The main implementation technologies of XOffsetDatastructure are presented in this section.

5.1 Custom Allocator

- Allocator
- According to Wikipedia:
  - 'In C++ computer programming, allocators are a component of the C++ Standard Library. Allocators handle all the requests for allocation and deallocation of memory for a given container.'

5.1 Custom Allocator

- Allocator
- According to Wikipedia:
  - 'In C++ computer programming, allocators are a component of the C++ Standard Library. Allocators handle all the requests for allocation and deallocation of memory for a given container.'
- Pablo Halpern, in his presentation 'Allocators, the Good Parts', offers a simpler definition:
  - 'An allocator is an object that provides the following two basic services:
  - Allocate – return a specified amount correctly-aligned memory for use by the client.
  - Deallocate – return the specified memory to the allocator for eventual reuse.'

5.1 Custom Allocator

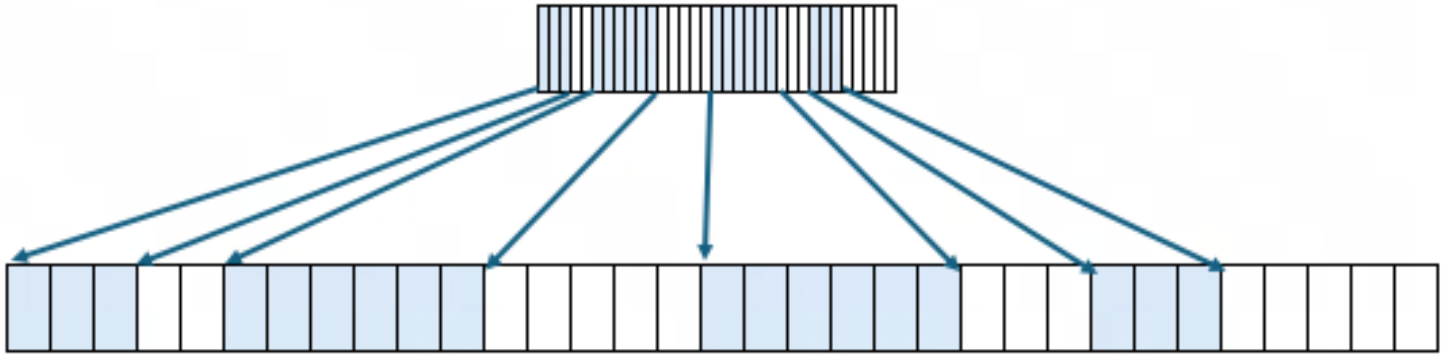
- In our XOffsetDatastructure implementation, we've chosen to use a **Custom Allocator** for two primary reasons:
- Performance Improvement:
  - Our custom allocator is designed to provide **faster allocations and deallocations** compared to standard allocators.
  - Additionally, it aims to improve **memory locality**, which can lead to better cache utilization and overall performance gains.

5.1 Custom Allocator

- In our XOffsetDatastructure implementation, we've chosen to use a **Custom Allocator** for two primary reasons:
- Performance Improvement:
  - Our custom allocator is designed to provide **faster allocations and deallocations** compared to standard allocators.
  - Additionally, it aims to improve **memory locality**, which can lead to better cache utilization and overall performance gains.
- Ensuring Contiguous Memory:
  - A crucial aspect of our XOffsetDatastructure approach is maintaining data in **contiguous memory**.
  - Our custom allocator is specifically designed to guarantee this contiguous memory layout, which is essential for our zero-encoding and zero-decoding strategy.

5.1 Custom Allocator

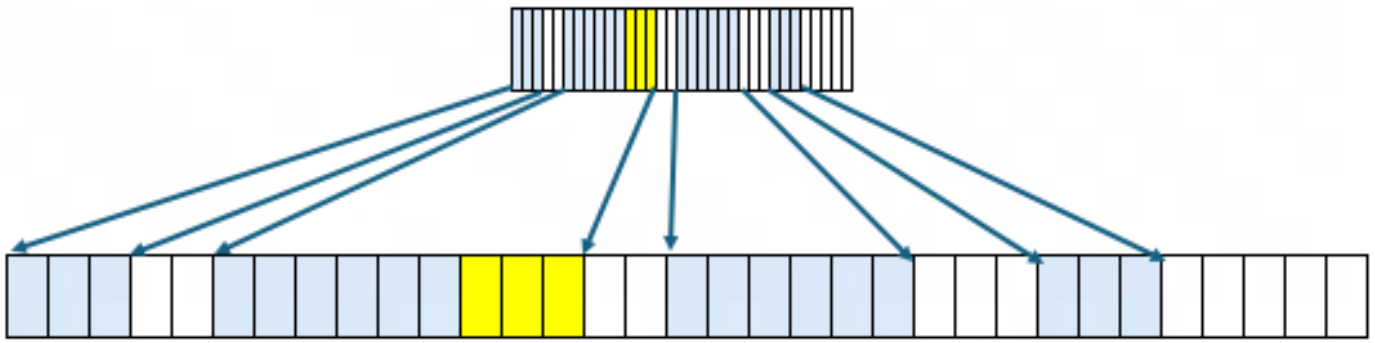
- Our Custom Allocator is based on a **bitmap** management system.
- The XOffsetDatastructure maintains a block of memory space with contiguous virtual addresses.
- This memory block is divided into **chunks**, and we use a bitmap to track the usage status of each chunk.



Bitmap Allocator

5.1 Custom Allocator

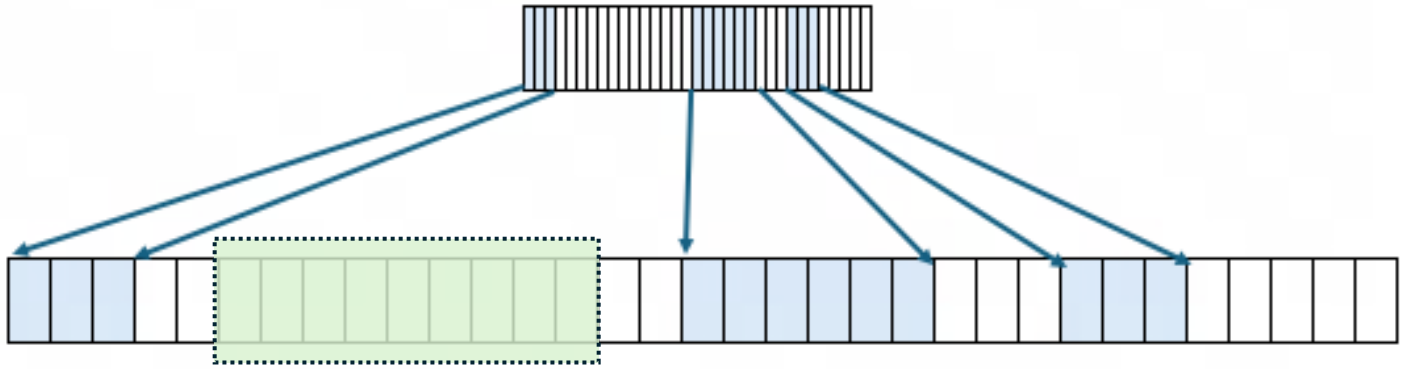
- mallocN
  - This function is used to find N consecutive free chunks.
- Built-in Functions
  - The implementation is optimized using built-in functions to accelerate the search process.
  - In MSVC, we use `_BitScanForward64/_BitScanReverse64`. In GCC, we use `__builtin_ctzll`. In Clang, we use `__builtin_ffsll`.
- As shown in the diagram, we've used mallocN to allocate a block of 3 contiguous chunks.



Bitmap Allocator (After mallocN)

5.1 Custom Allocator

- freeN
  - This function is used to release N consecutive chunks of memory.
- The implementation of freeN is designed to efficiently mark these chunks as available in our bitmap.
- The diagram illustrates how we use freeN to release a block of memory, making it available for future allocations.



Bitmap Allocator (After freeN)

5.1 Custom Allocator

- This bitmap-based approach, combined with the use of optimized built-in functions, allows our **Custom Allocator** to perform rapid allocations and deallocations while maintaining a contiguous memory layout.
- It can, of course, be replaced by other efficient allocators.

5.1 Custom Allocator

- This bitmap-based approach, combined with the use of optimized built-in functions, allows our **Custom Allocator** to perform rapid allocations and deallocations while maintaining a contiguous memory layout.
- It can, of course, be replaced by other efficient allocators.
- For those interested in further **optimizing bitmap allocators**, there are several avenues to explore:
  - Fast Bitmap Fit: A CPU Cache Line friendly memory allocator for single object allocations
  - Bit allocator based on segment tree algorithm.
- For a broader perspective on **allocator**, there's a wealth of resources available from recent CppCon:
  - CppCon 22 Understanding Allocator Impact on Runtime Performance by Parsa Amini
  - CppCon 20 Practical memory pool based allocators for Modern C++ by Misha Shalem
  - CppCon 19 Getting Allocators out of Our Way by Alisdair Meredith & Pablo Halpern
  - CppCon 18 An Allocator is a Handle to a Heap by Arthur O'Dwyer
  - CppCon 18 Fancy Pointers for Fun and Profit by Bob Steagall
  - CppCon17 Allocators, the Good Parts by Pablo Halpern
  - CppCon 17 From Security to Performance to GPU Programming - Exploring Modern Allocators by Sergey Zubkov
  - CppCon 17 How to Write a Custom Allocator by Bob Steagall
  - CppCon 16 Lightweight Object Persistence With Relocatable Heaps in Modern C++ by Bob Steagall

5.2 Offset Pointers

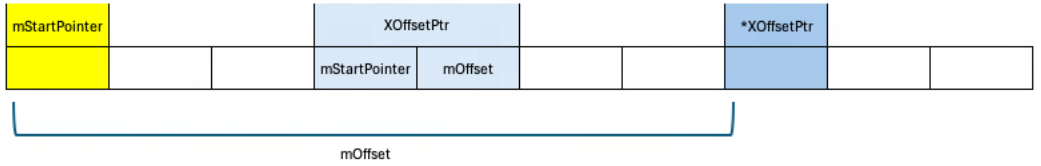
- Absolute Pointers and Relative/Offset Pointers
  - Pointers can be categorized into two main types: **Absolute pointers** and **Relative/Offset pointers**.
  - Absolute pointers represent the actual address in virtual memory.
  - Relative or Offset pointers represent an offset from an absolute start address (often called the "base").

5.2 Offset Pointers

- Absolute Pointers and Relative/Offset Pointers
  - Pointers can be categorized into two main types: **Absolute pointers** and **Relative/Offset pointers**.
  - Absolute pointers represent the actual address in virtual memory.
  - Relative or Offset pointers represent an offset from an absolute start address (often called the "base").
- Smaller and Movable
  - The key advantages of offset pointers are: **Smaller** and **movable**.
  - They are smaller, requiring less storage space.
  - They are movable, meaning the data structure can be relocated in memory without invalidating the pointers.

5.2 Offset Pointers

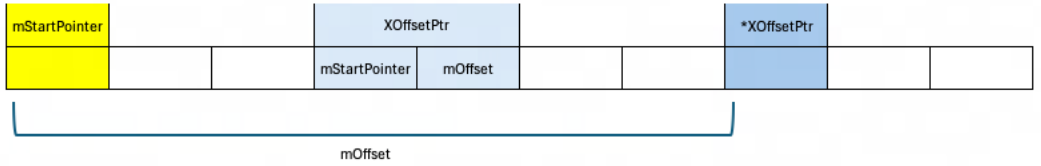
- In our implementation, we use two types of offset pointers:
- 1. **Offset from a custom base address**:
- As shown in the diagram, this type of offset pointer stores a value relative to a predefined base address. To resolve the actual address, we add this offset to the base address.



Offset Pointer 1

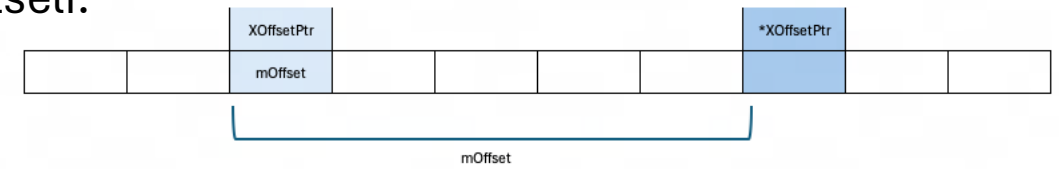
5.2 Offset Pointers

- In our implementation, we use two types of offset pointers:
- 1. **Offset from a custom base address**:
- As shown in the diagram, this type of offset pointer stores a value relative to a predefined base address. To resolve the actual address, we add this offset to the base address.



Offset Pointer 1

- 2. **Offset from 'this' pointer**:
- For this type, we directly use 'boost::interprocess::offset_ptr'. As illustrated in the diagram, this offset pointer stores a value relative to its own location. To resolve the actual address, we add this offset to the address of the pointer itself.



Offset Pointer 2

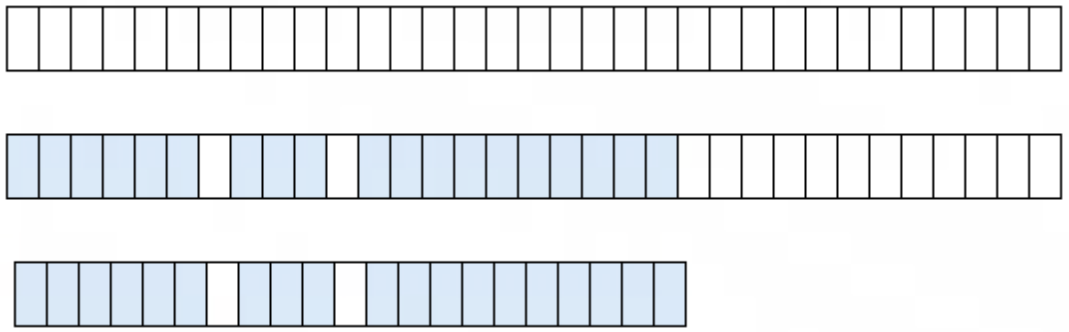
5.3 Containers with Offset Pointers

- Boost provides several containers that **support custom pointer** types. This means we can use these containers directly in our XOffsetDatastructure implementation.
- By using these containers, we can **avoid** some of the **inconsistencies** that may arise from different STL implementations across various compilers and platforms.

```
using XString = boost::container::basic_string<char, std::char_traits<char>, XSimpleAllocator<char, XSimpleStorage>>;
template <typename T>
using XVector = boost::container::vector<T, XSimpleAllocator<T, XSimpleStorage>>;
template <typename T>
using XFlatSet = boost::container::flat_set<T, std::less<T>, XSimpleAllocator<T, XSimpleStorage>>;
template <typename T>
using XSet = boost::container::set<T, std::less<T>, XSimpleAllocator<T, XSimpleStorage>>;
template <typename K, typename V>
using XFlatMap = boost::container::flat_map<K, V, std::less<K>, XSimpleAllocator<std::pair<K, V>, XSimpleStorage>>;
template <typename K, typename V>
using XMap = boost::container::map<K, V, std::less<K>, XSimpleAllocator<std::pair<const K, V>, XSimpleStorage>>;
```

5.4 Auto-Resizing Mechanisms

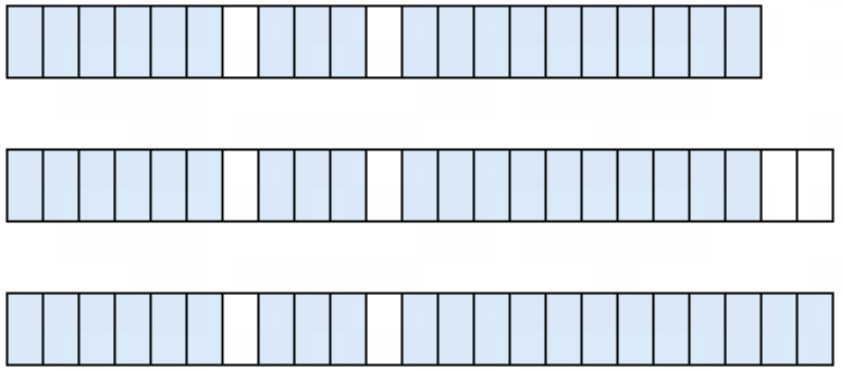
- We've implemented two main approaches for managing memory allocation in our XOffsetDatastructure:
- 1. **Pre-allocation With Trimming**:
  - In this approach, we start by allocating a larger block of memory than we might need.
  - For example, as shown in the diagram, we might initially allocate 4096 bytes.
  - After writing our data, we can call a 'trim' function to release any unused chunks back to the system.
  - This method can be efficient when we have a good estimate of our maximum memory needs.



Per-allocation with Trimming

5.4 Auto-Resizing Mechanisms

- 2. **Auto-Resizing**:
- For greater ease of use, we've also implemented auto-resizing mechanisms. These automatically trigger memory expansion when the current space is insufficient for a new operation.
- Here's an example of how this works in code: RETRY_IF_BAD_ALLOC(r1->mSet.insert(4), holder);
- As illustrated in the diagram, when there's not enough space for an operation, the system automatically triggers a resize. It's important to note that this may cause the entire buffer to be moved in memory.



Auto-Resizing

5.4 Auto-Resizing Mechanisms

- We've implemented the auto-resizing functionality using three different methods:
  - std::exception, longjmp and Boost::LEAF.

Method	Time (ms)
Exception	1735
longjmp	1395
Leaf	3177

Performance Comparison

5.4 Auto-Resizing Mechanisms

- We've implemented the auto-resizing functionality using three different methods:
  - std::exception, longjmp and Boost::LEAF.

Method	Time (ms)
Exception	1735
longjmp	1395
Leaf	3177

Performance Comparison

- Pre-Allocation with Trimming and Auto-Resizing
  - The choice between pre-allocation with trimming and auto-resizing, as well as the specific auto-resizing method, can be made based on the specific requirements of the application.
  - Pre-allocation might be preferred in scenarios where memory usage is more predictable, while auto-resizing offers more flexibility for dynamic memory needs.

5.5 Summary

- Custom Allocator
  - Our custom allocator serves 2 purposes: Contiguous Memory and Performance Improvement.
- Offset Pointers
  - We've implemented offset pointers which offer two main advantages: Smaller and Movable.
- Containers with Offset Pointers
  - By leveraging Boost containers that support custom pointer types, we're able to create and manage complex data structures while maintaining the benefits of the offset-based approach.
- Auto-resizing Mechanisms:
  - We've implemented two approaches: Pre-allocation with trimming and Auto-resizing.

5.5 Summary

- Custom Allocator
  - Our custom allocator serves 2 purposes: Contiguous Memory and Performance Improvement.
- Offset Pointers
  - We've implemented offset pointers which offer two main advantages: Smaller and Movable.
- Containers with Offset Pointers
  - By leveraging Boost containers that support custom pointer types, we're able to create and manage complex data structures while maintaining the benefits of the offset-based approach.
- Auto-resizing Mechanisms:
  - We've implemented two approaches: Pre-allocation with trimming and Auto-resizing.
- These 4 features, along with others in the codebase, form the foundation of XOffsetDatastructure.
- https://github.com/ximicpp/XOffsetDatastructure

6. Limitations and Plans

The limitations and plans of XOffsetDatastructure are presented in this section.

6.1 Memory Usage

- To assess our **current memory usage**, we conducted a test using **10,000** sample **Units**.
- The results of this test are as follows:
  - When using standard STL containers, the memory usage was **134MB**.
  - With our current implementation of XOffsetDatastructure, the memory usage was **178MB**.
- These results indicate that our current implementation of XOffsetDatastructure is using more memory than the standard STL containers.

Algorithms(growth factor, prealloc)	Memory Usage (10 thousand units, MB)
std::*	134
XOffsetDatastructure(60, false)	178

Memory Usage

6.1 Memory Usage

- Growth Factor and Pre-Allocation
  - We're focusing on two main strategies to optimize memory usage: adjusting the Growth Factor and implementing Pre-Allocation.
- Growth Factor
  - As shown in the diagram, we're customizing the growth factor of our containers. This allows us to strike a balance between memory savings and access speed. By fine-tuning this factor, we can reduce unnecessary memory allocation while still maintaining efficient performance.

```
struct growth_factor_custom : boost::container::dtl::grow_factor_ratio<0, 16, 10> {};

using vector_option = boost::container::vector_options_t<
boost::container::growth_factor<growth_factor_custom> >;

template <typename T> using XVector = boost::container::vector<T,
XOffsetDatastructure::XSimpleAllocator<T, XSimpleStorage>, vector_option>;
```

6.1 Memory Usage

- Pre-Allocation
  - Instead of adding data to a vector one by one, we can pre-set the vector's size. The example provided illustrates how we can use pre-allocation.
  - This approach can reduce the number of reallocations needed as the vector grows, potentially saving both memory and time.

```
rootptr->path.resize(16);
for (auto i = 0; i < 16; ++i)
{
    rootptr->path[i] = {disFloat(gen), disFloat(gen), disFloat(gen)};
}

for (auto i = 0; i < 16; ++i)
{
    rootptr->path.emplace_back(disFloat(gen), disFloat(gen), disFloat(gen));
}
```

6.1 Memory Usage

- Memory Usage and Access Time
  - let's examine the performance comparisons in two key areas: Memory Usage and Access Time.
- Memory Usage
  - As the table illustrates, through our optimizations with growth factor and pre-allocation, we've managed to bring the memory usage of XOffsetDatastructure in line with STL performance, and in some cases, even surpass it.

Algorithms(growth factor, prealloc)	Memory Usage (10 thousand units, MB)
std::*	134
XOffsetDatastructure(60, false)	178
XOffsetDatastructure(60, true)	158
XOffsetDatastructure(10, false)	140
XOffsetDatastructure(10, true)	124

Memory Usage

6.1 Memory Usage

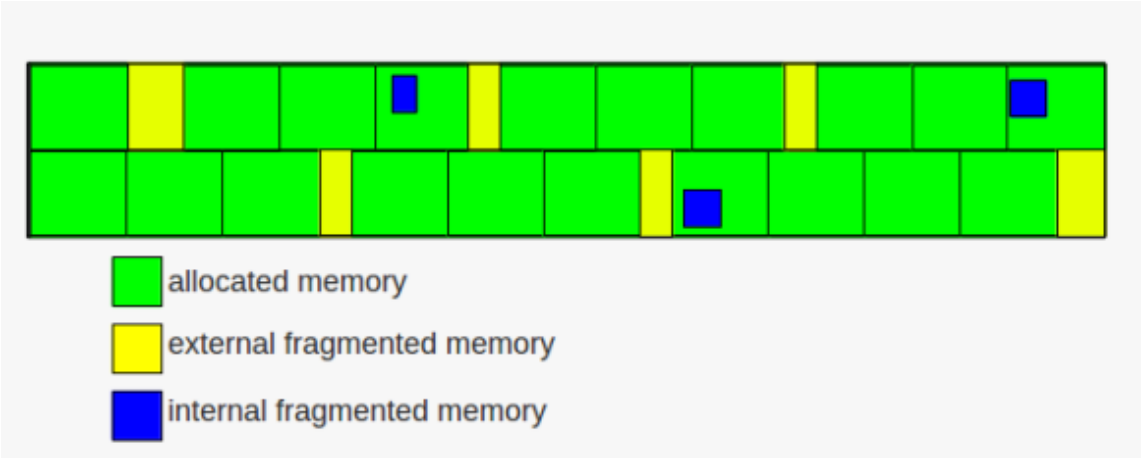
- Memory Usage and Access Time
- Access Time
- The table comparing access times shows that there‘s no significant difference in access times across different parameters (Growth Factor).

Growth Factor	Read/Write	Encoding/Decoding
XOffsetDatastructure(10)	51.4463ms	0.02936ms
XOffsetDatastructure(60)	51.8268ms	0.02996ms

Performance Comparision

6.2 Memory Fragmentation

- Memory Fragmentation
- As our previous optimizations have shown, there's always room for improvement in our custom allocator.
- One area we've identified for further enhancement is addressing **memory fragmentation**. Our current implementation exhibits both **intra-chunk** (or internal) and **inter-chunk** (or external) fragmentation.
- As illustrated in the diagram, our current allocator design is prone to both these types of fragmentation.



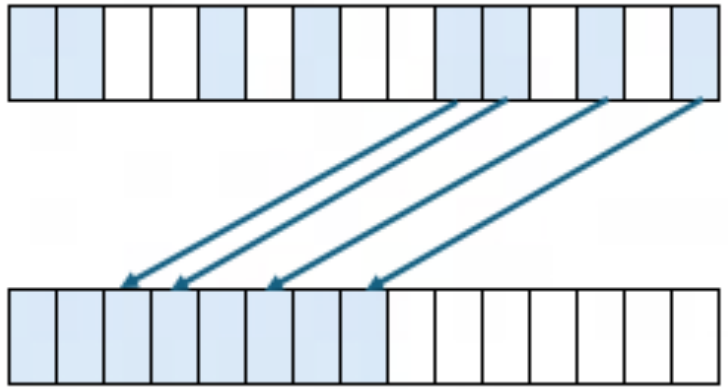
Memory Fragmentation

6.2 Memory Fragmentation

- The fragmentation issue can be significantly mitigated through improvements in **allocator design**.
- Allocator design is an active area of research in computer science, receiving considerable attention and **support** from the academic and industry communities. This ongoing research provides us with a wealth of potential solutions and improvements to explore.
  - Randomized C/C++ dynamic memory allocator
  - VCMalloc: A Virtually Contiguous Memory Allocator
  - Learning-based Memory Allocation for C++ Server Workloads
  - …
  - CppCon 22 Understanding Allocator Impact on Runtime Performance by Parsa Amini
  - CppCon 20 Practical Memory Pool Based Allocators for Modern C++ by Misha Shalem
  - CppCon19 Getting Allocators out of Our Way by Alisdair Meredith & Pablo Halpern
  - …
- How to improve the custom allocator is a topic that will be continued.

6.2 Memory Fragmentation

- A **compact** mechanism, also known as memory compaction, is an operation that reorganizes memory to eliminate fragmentation.
- As illustrated in the diagram, a compact operation rearranges allocated memory blocks, moving them together to create larger contiguous free spaces.
- How to improve the **efficiency** of the compact mechanism is a direction we will continue to explore.



Memory Compaction

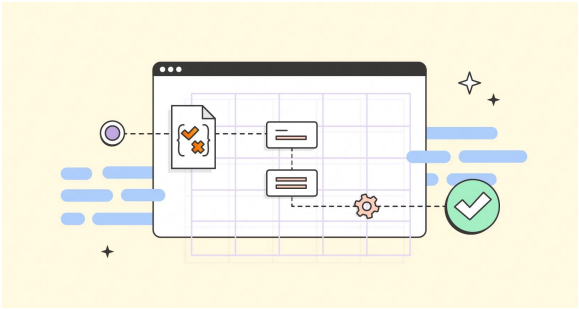
6.3 Compatibility

- Compatible Subset
- As we mentioned earlier, **compatibility** was initially a lower priority for us as we focused on optimizing **performance**.
  - This approach was justified because in our primary application scenarios, the compiler, platform, architecture, and language remain consistent between servers.
- However, we've created a **compatible subset** of our data structures. This subset is designed to work consistently across different platforms and compilers.
- As demonstrated in the code example, the types shown in our demo are compatible across different platforms:
  - clang/msvc/gcc
  - macOS/Windows/Linux

```
class TestType
{
public:
   int mInt{0};
   float mFloat{0.f};
   XVector<int> mVector;
   XVector<XString> mStringVector;
   class TestTypeInner
   {
   public:
      int mInt{0};
      XVector<int> mVector;
   } TestTypeInnerObj;
   XVector<TestTypeInner> mXXTypeVector;
   XMap<XString, TestTypeInner> mComplexMap;
   XSet<XString> mStringSet;
   XSet<int> mSet;
   XString mString;
};
```

6.4 Validation

- Schema Validation
  - It involves performing proper data validation before ingesting an XOffsetDatastructure buffer.
  - Typically, this is done using schema validation techniques when dealing with structured data.
- Out of Range Safety
  - Another feature we're considering is ensuring that reading an XOffsetDatastructure buffer does not access any memory outside the original buffer.
- Other Data Validation Features



Data Validation

7. Summary and Takeaways

The summary and takeaways are presented in this section.

7.1 Summary

- Zero-Encoding and Zero-Decoding
  - XOffsetDatastructure is a serialization library designed to reduce or even eliminate the performance consumption of **serialization** and **deserialization** by utilizing zero-encoding and zero-decoding.
- High-Performance Read and Write
  - XOffsetDatastructure is also a collection of **high-performance** data structures designed for efficient **read** and in-place/non-in-place **write**, with performance comparable to STL.

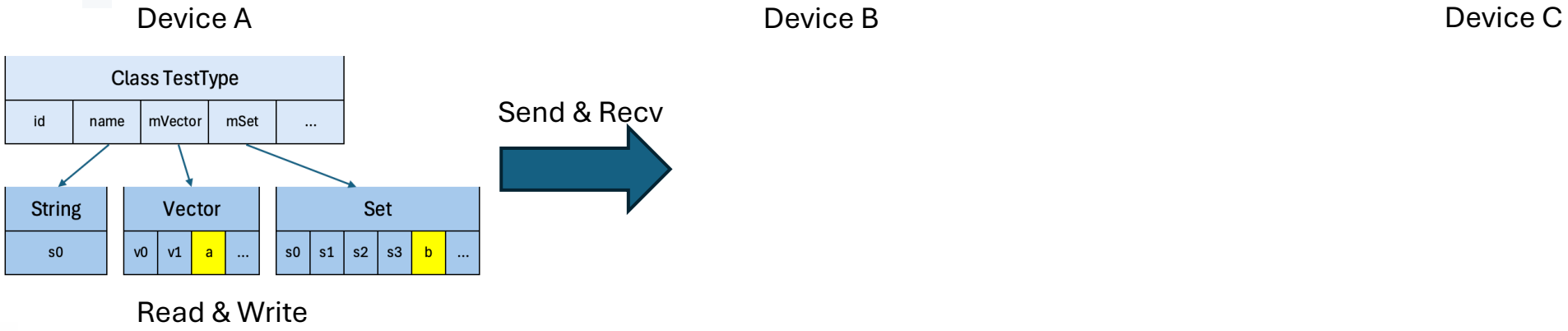
7.1 Summary

- Zero-Encoding and Zero-Decoding
  - XOffsetDatastructure is a serialization library designed to reduce or even eliminate the performance consumption of **serialization** and **deserialization** by utilizing zero-encoding and zero-decoding.
- High-Performance Read and Write
  - XOffsetDatastructure is also a collection of **high-performance** data structures designed for efficient **read** and in-place/non-in-place **write**, with performance comparable to STL.
- XOffsetDatastructure offers a powerful solution for applications that require both fast data manipulation and quick data transfer or storage, such as in high-performance computing, game development, or real-time data processing systems.

7.2 Takeaways

- This code snippet demonstrates the lifecycle of data using XOffsetDatastructure across three different processes or devices (A, B, and C).

```
// process/device A
XTypeHolder<TestType> holder;
holder.mVector.push_back(a); // non-in-place write
holder.mSet.insert(b); // non-in-place write
std::vector<std::byte> data = holder.getBuffer(); // zero-encoding
...
```

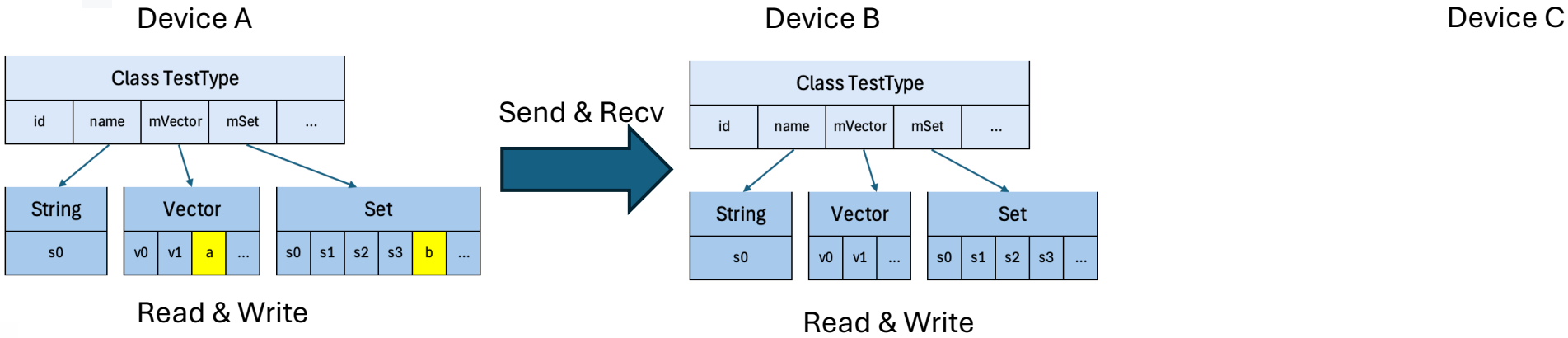


Data Flow of XOffsetDatastructure Across Processes/Devices

7.2 Takeaways

- This code snippet demonstrates the lifecycle of data using XOffsetDatastructure across three different processes or devices (A, B, and C).

```
// process/device B
XTypeHolder<TestType> holder(data); // zero-decoding
holder.mVector.pop_back(); // non-in-place write
holder.mSet.erase(b); // non-in-place write
std::vector<std::byte> data = holder.getBuffer(); // zero-encoding
...
```

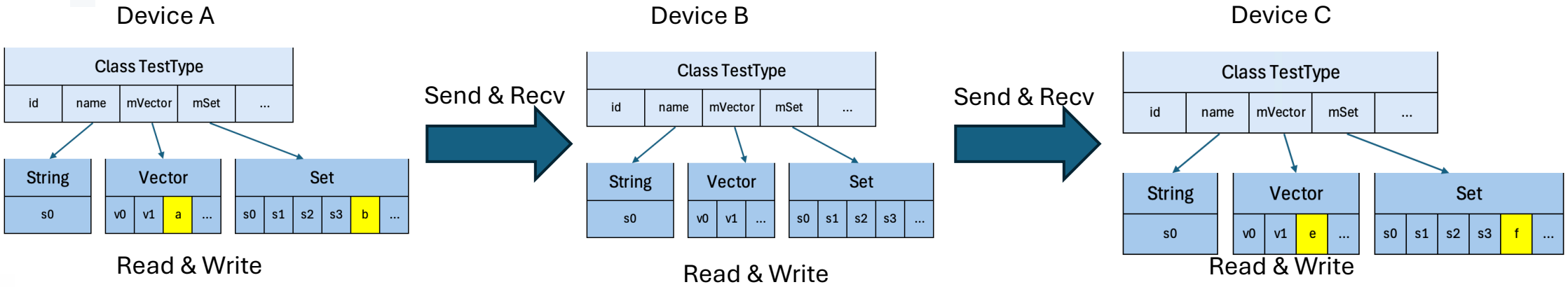


Data Flow of XOffsetDatastructure Across Processes/Devices

7.2 Takeaways

- This code snippet demonstrates the lifecycle of data using XOffsetDatastructure across three different processes or devices (A, B, and C).

```
// process/device C
XTypeHolder<TestType> holder(data); // zero-decoding
holder.mVector.push_back(e); // non-in-place write
holder.mSet.insert(f); // non-in-place write
std::vector<std::byte> data = holder.getBuffer(); // zero-encoding
...
```



Data Flow of XOffsetDatastructure Across Processes/Devices

Thank you!

Any questions?