



# Lecture 3: Storage Management





# Logistics

- Difference between 4420 and 6422 sections
  - Advanced lectures
  - Advanced questions in exams
- Assignment 1 released on Gradescope



# Recap

- Tour of relational operators
- BuzzDB
- Why C++?



# Lecture Overview

- Storage Management
- Tuple and Field classes
- Generalized Tuple class
- Smart pointers

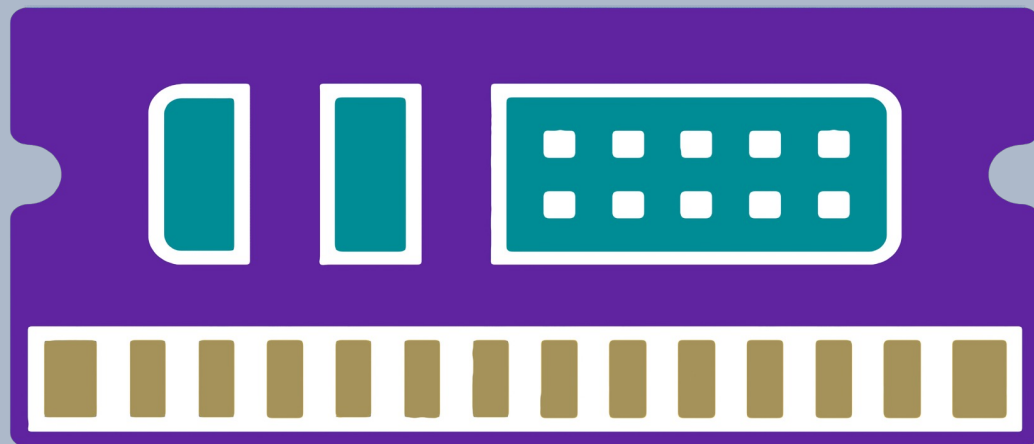


# Storage Technologies

Storage  
Technologies

Persistent  
Device

## VOLATILE STORAGE



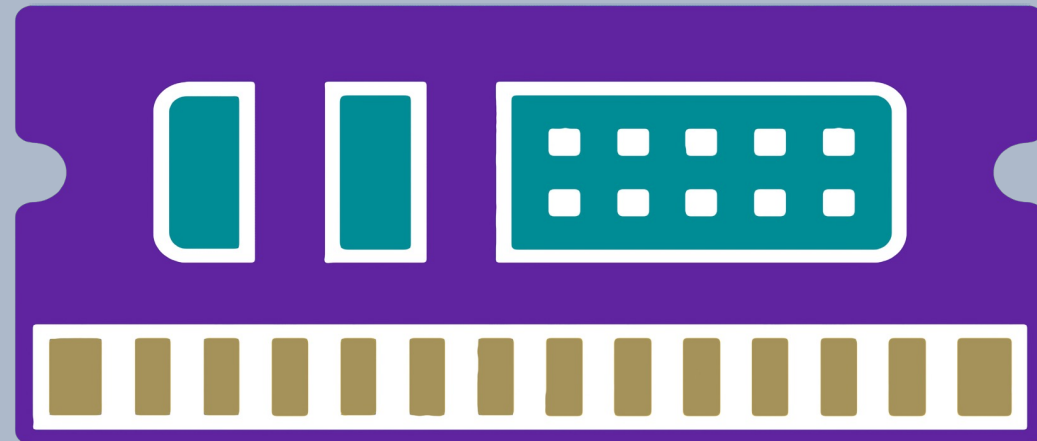
## PERSISTENT STORAGE





# Volatile Storage

## VOLATILE DRAM



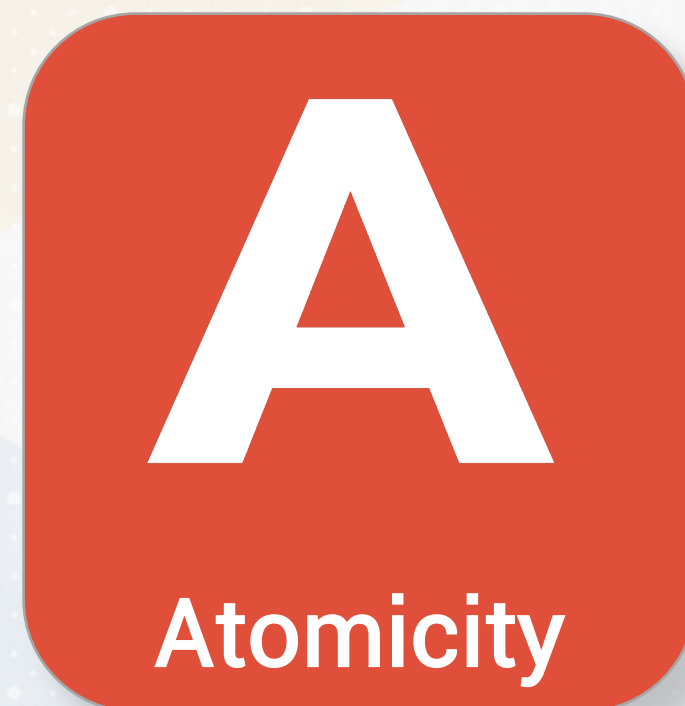
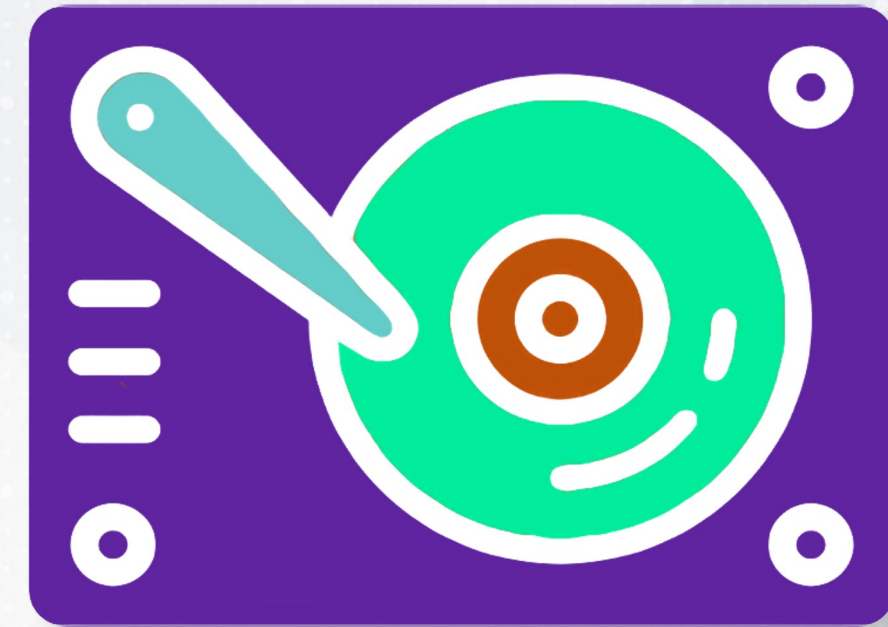
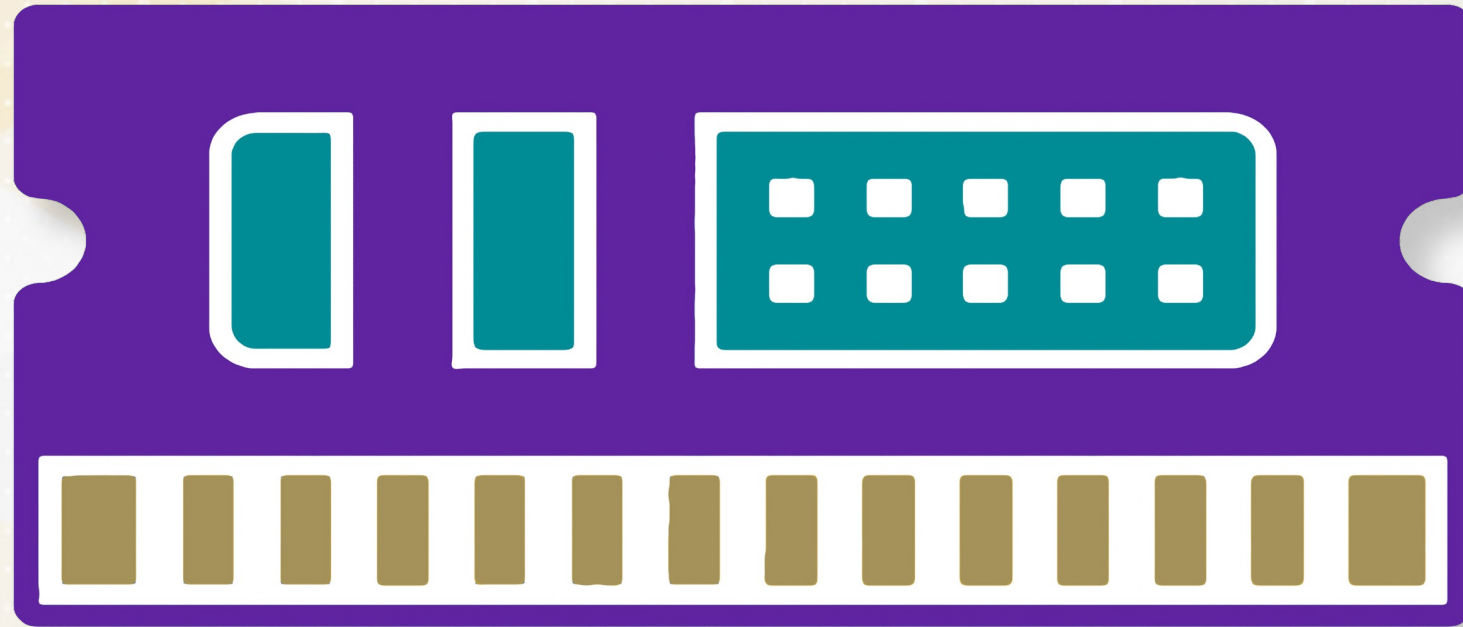
DRAM  
(Dynamic Random-Access Memory)

Ideal for Quick Data Access

Data Lost Without Power

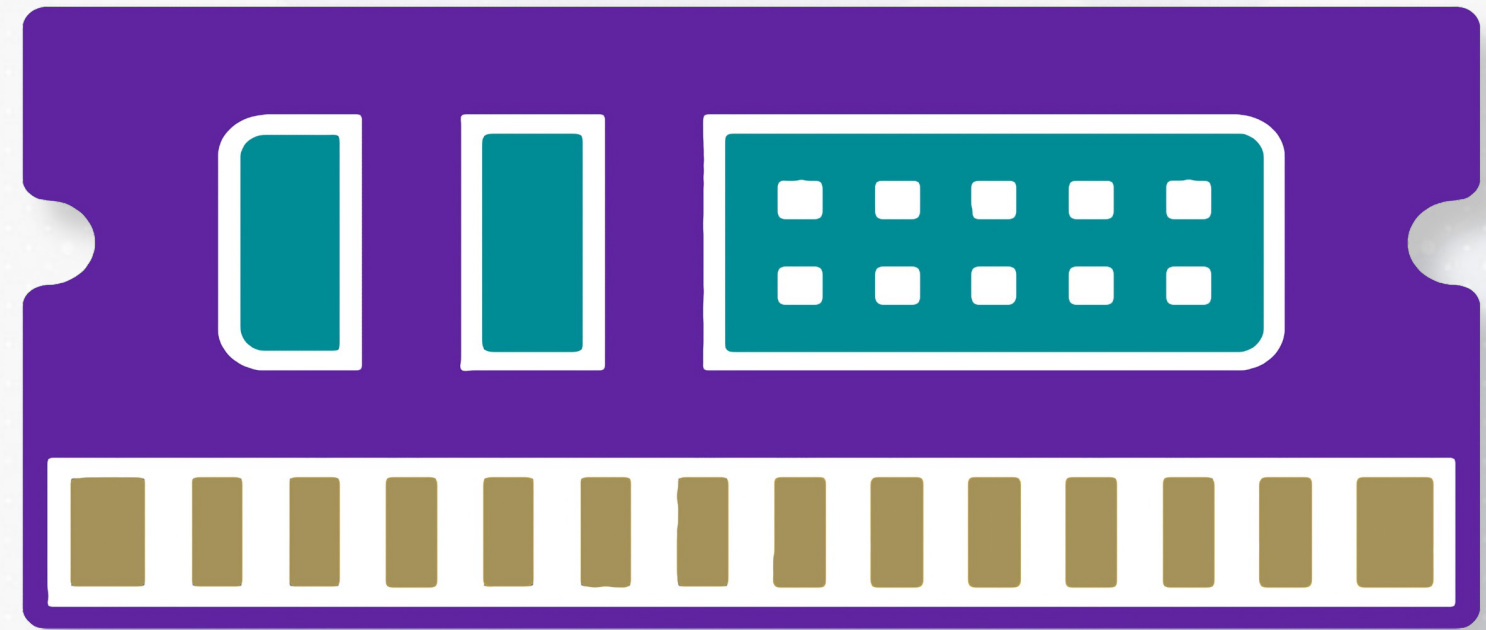


# Persistent Storage





# Data Lifecycle: Stage One



"Hot Data" Storage

es &  
es



# Data Lifecycle: Stage Two

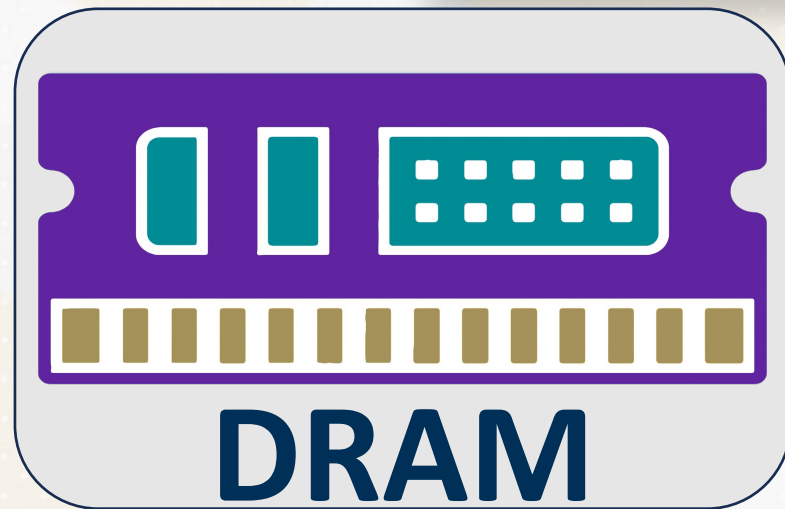


Speed & Durability Secure



# Data Lifecycle

*Faster access - not durable*



|  |  |  |
|--|--|--|
|  |  |  |
|  |  |  |
|  |  |  |

*Cached Pages*



|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

*Database*

*Slower access - but durable*



# FILE I/O IN C++





# File I/O in C++

Read/Write  
Data



DRAM



# File I/O in C++

**buzzDB**



```
int main() {
    BuzzDB db;
    // Import data from "output.txt" file
    std::ifstream inputFile("output.txt");
    // Attempt to open file
    int field1, field2;
    // Read pairs of integers from the file and insert them into
    BuzzDB
    while (inputFile >> field1 >> field2) {
        db.insert(field1, field2);
    }
    // Perform aggregation query on the imported data
    db.selectGroupBySum();
}
```



# File I/O in C++

Verify File  
Availability

Use std::cerr  
Stream

Avoids Data  
Disruption

```
std::ifstream inputFile("output.txt"); // Attempt to open file

if (!inputFile) {
    std::cerr << "Unable to open file" << std::endl; // Error handling
    return 1;
}
```



# Query Performance in C++

## C++ Chrono Library Efficiency

```
// Get the start time
auto start = std::chrono::high_resolution_clock::now();
BuzzDB db;
...
db.selectGroupBySum();
// Get the end time
auto end = std::chrono::high_resolution_clock::now();
// Calculate the difference between end and start times
std::chrono::duration<double> elapsed = end - start;
// Output the elapsed time in seconds
std::cout << "Elapsed time: " << elapsed.count() << " seconds" <<
std::endl;
```



# Why does timing matter?



Optimizes Database Operations

Benchmark BuzzDB  
Performance



# Tuple and Field Classes





# Tuple Class

Integer Key-  
Value Pairs

Strings

```
class Tuple {  
public:  
    // Identifier field  
    int key;  
    // Actual data field  
    int value;  
};
```



# Field Class

Field Class

Type

FieldType

INT &  
FLOAT

Enums

Defines  
Constants

```
enum FieldType {INT, FLOAT}  
  
class Field {  
    public:  
    FieldType type;  
    union {  
        int i;  
        float f;  
    } data;  
};
```



# Field Class

Field Class

Integer I  
Float F

Unions

Memory  
Share

Field Object

Memory  
Allocation

INT & FLOAT

4 Bytes  
Storage

```
enum FieldType {INT, FLOAT}

class Field {
public:
    FieldType type;
    union {
        int i;
        float f;
    } data;
};
```



# Field Class

Integer  
Value INT

Float Value  
FLOAT

Constructor  
Overloading

Print Method

```
class Field {  
    public:  
    Field(int i): type(INT), data{i} {}  
    Field(float f): type(FLOAT), data{f} {}  
    void print() const {  
        switch(type) {  
            case INT: std::cout << data.i;  
            break;  
            case FLOAT: std::cout << data.f;  
            break;  
        }  
    }  
}
```

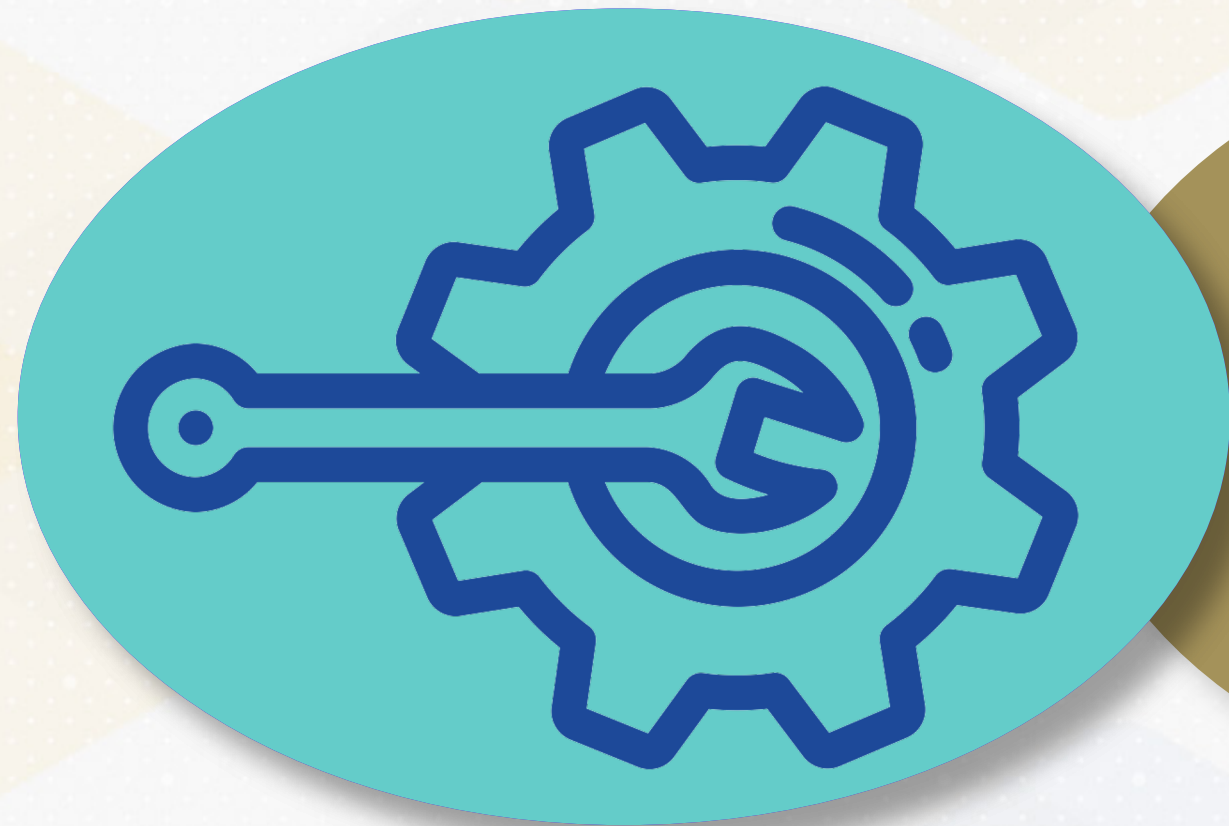


# C++ Constructors, Tuples, and Strings





# Constructors in C++



Initializes Integer  
& Float Data



# Generalized Tuple Class

| Field Value  |  |  |  |  |
|--------------|--|--|--|--|
| Integer      |  |  |  |  |
| Float Column |  |  |  |  |

```
class Tuple {
    std::vector<Field> fields;
public:
    void addField(const Field & field) {
        fields.push_back(field);
    }
    void print() const {
        for (const auto & field: fields) {
            field.print(); std::cout << " ";
        }
    }
};
```



# Constructing Generalized Tuples

```
void BuzzDB::insert(int key, int value) {  
    Tuple newTuple;  
    Field key_field = Field(key);  
    Field value_field = Field(value);  
    float float_val = 132.04;  
    Field float_field = Field(float_val);  
  
    newTuple.addField(key_field);  
    newTuple.addField(value_field);  
    newTuple.addField(float_field);  
  
    table.push_back(newTuple);  
    index[key].push_back(value);  
}
```



# Further Generalization to Strings

FieldType enum includes STRING

```
enum FieldType {INT, FLOAT, STRING}

class Field {
    public:
        FieldType type;
        union {
            int i;
            float f;
            char *s; // string
        } data;
        ...
}
```



# More Generalization to Strings

Polymorphic Container

Field Object

Floating  
Point

String  
Form

DATABASE = 8 Letters

POLYMORPHISM = 12  
Letters

# More Generalization to Strings

## Dynamic Memory Allocation

String  
Parameter S

Field Object  
to String

String S +  
1

Memory  
data.s

```
Field(const std::string &s) : type(STRING) {  
    data.s = new char[s.size() + 1];  
    std::copy(s.begin(), s.end(), data.s);  
    data.s[s.size()] = '\\0'; // Null-terminate the string  
}
```



# New Operator



Dynamic  
Memory  
Allocation

Heap-  
Allocated  
Variable

Explicit  
Variable  
Destruction

```
data.s = new char[s.size() + 1];
```

# Destructors in C++



Field Class  
Destructor

Cleans Memory

Data.s  $\neq$  nullptr

```
Field::~~Field() {  
    if (type == STRING && data.s != nullptr) {  
        delete[] data.s;  
    }  
}
```



# Delete Operator



Heap Memory  
Clearance

Correspondin  
g Delete  
Operator

Avoid  
System  
Crashes

```
delete ptr;    // Frees memory allocated for a single integer  
delete[] arr; // Frees memory allocated for an array of integers
```



# Raw Pointers





# Raw Pointers

Field Class + Raw Character Pointer

```
enum FieldType {INT, FLOAT, STRING}

class Field {
    public:
        FieldType type;
        union {
            int i;
            float f;
            char *s;
        } data;
        ...
};
```

# Perils of Raw Pointers

## Four Important Problems

Memory  
Leaks

Dangling  
Pointers

Double-free  
Errors

Ownership  
Ambiguity



# Perils of Raw Pointers: Memory Leaks

Explicit Deletion Required

```
void createMemoryLeak() {  
    int *leakyPtr = new int(42); // Allocation  
    // Forgot to delete leakyPtr  
    // Memory allocated to leakyPtr is never freed  
}
```

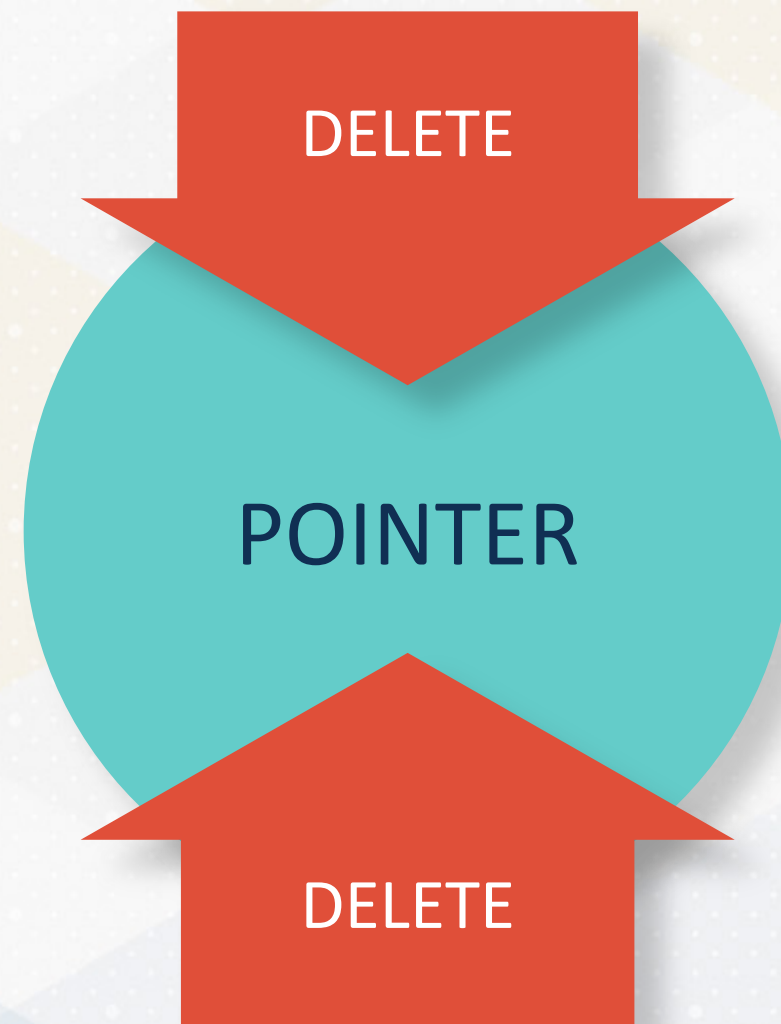
# Perils of Raw Pointers: Dangling Pointers



```
void createDanglingPointer() {  
    int * danglyPtr = new int(42)  
    delete danglyPtr  
    // Deallocate memory  
    // danglyPtr now becomes a dangling pointer  
    std::cout << *danglyPtr  
    // Undefined behavior, potentially a crash  
}
```



# Perils of Raw Pointers: Double-Free Errors



```
void createDoubleFree() {  
    int *doubleTroublePtr = new int(42);  
    delete doubleTroublePtr; // Correct deletion  
    delete doubleTroublePtr; // Double free, leads to runtime error  
}
```

# Perils of Raw Pointers

No Clear Ownership

Unclear Deallocation

Unclear Memory  
Management

```
int *function1(int *ptr) {  
    // It's unclear if the function should delete ptr  
    // Function logic here...  
    return new int(55); // Should the caller delete the returned pointer?  
}  
void function2() {  
    int *myPtr = new int(42);  
    int *newPtr = function1(myPtr);  
    // Who owns myPtr and newPtr? Who is responsible for deleting them?  
}
```



# Smart Pointers





# Smart Pointers

`std::unique_ptr`

Code Safety

Readability

Maintainability





# Smart Pointers: Avoid Memory Leaks



```
#include <memory>

void createMemoryLeak() {
    std::unique_ptr<int> safePtr =
        std::make_unique<int>(42); // Automatic management
                                   // No need to manually delete;
    // memory is automatically freed when safePtr goes out of scope
}
```

`std::unique_ptr`

Manages Memory

Scope Deallocation

# Smart Pointers: Avoid Dangling Pointers

After calling `reset()`, `safePtr` safely releases its ownership and avoids becoming a dangling pointer by setting itself to `nullptr`

```
void createDanglingPointer() {  
    std::unique_ptr<int> safePtr = std::make_unique<int>(42);  
    safePtr.reset(); // Correctly frees memory and sets pointer to nullptr  
    // safePtr is now nullptr, preventing access to freed memory  
}
```



# Smart Pointers: Avoid Double-Free Errors

`std::unique_ptr` enforces unique ownership of the managed object, meaning that the memory is freed exactly once when the pointer goes out of scope or is reset.

Prevents  
double-free errors

```
void createDoubleFree() {  
    std::unique_ptr<int> safePtr = std::make_unique<int>(42);  
    // safePtr goes out of scope and automatically deletes the managed object  
    // No risk of double free errors  
    // as unique_ptr ensures single ownership and controlled deletion  
}
```

# Smart Pointers: Avoid Ownership Ambiguity

unique\_ptr clarifies ownership through its ownership model

```
std::unique_ptr<int> function1(std::unique_ptr<int> ptr) {  
    // Ownership is clearly transferred with unique_ptr, no ambiguity  
    return std::make_unique<int>(55);  
    // Returning a new unique_ptr transfers ownership back to the caller  
}  
void function2() {  
    std::unique_ptr<int> myPtr = std::make_unique<int>(42);  
    std::unique_ptr<int> newPtr = function1(std::move(myPtr));  
    // Ownership transfer is explicit with std::move,  
    // clarifying lifecycle management  
}
```



# Conclusion

- Storage management
- Generalized Tuple class
- Smart Pointers