Lecture 4:
Smart Pointers and Pages


Logistics

- Point Solutions App
  - Session ID: **database**
- Assignment 1 released on Gradescope

Recap

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

Lecture Overview

- Smart Pointers
- Smart Field
- Heap vs Stack
- Page class
- Serialization

Smart Pointers

Smart Pointers

- std::unique_ptr
- Code Safety
- Readability
- Maintainability

Smart Pointers: Avoid Memory Leaks

```cpp
#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
}
```

Smart Field

Smart Pointer in Field

Transition from manual memory management to std::unique_ptr for string

```
class Field {
public:
    int data_i;
    float data_f;
    std::unique_ptr <char[]> data_s;
    size_t data_s_length;
    ...
}
```

Smart Pointer in Field

unique_ptr

String Management Leak Risk Minimized

Destructor Eliminates delete[] Calls

```
Field(const std::string &s) : type(STRING) {
  data_s_length = s.size() + 1;
  data_s = std::make_unique<char[]>(data_s_length);
  std::strcpy(data_s.get(), s.c_str());
}
```

Smart Tuple Class

Tuple Class

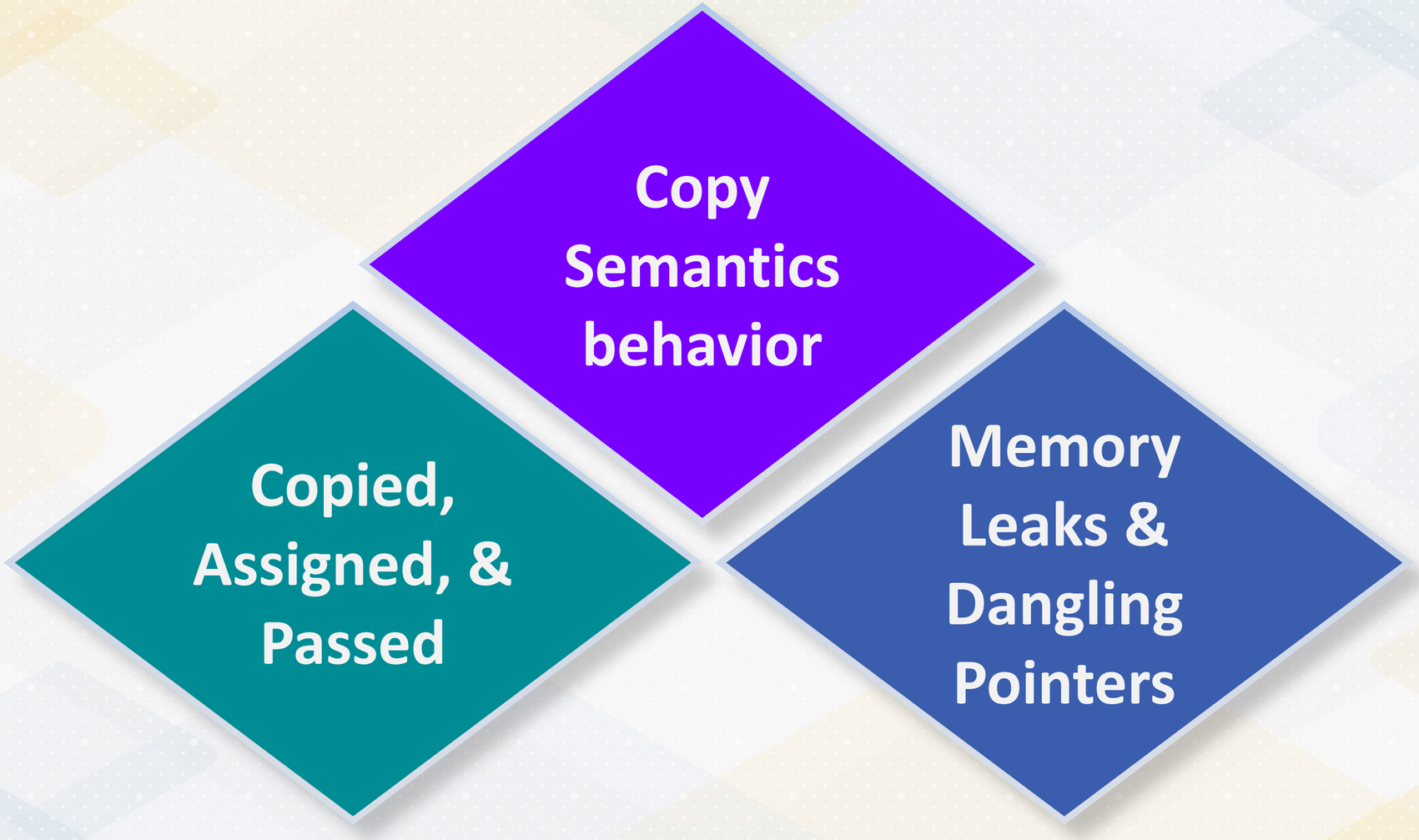
std::vector

std::unique_ptr<Field>

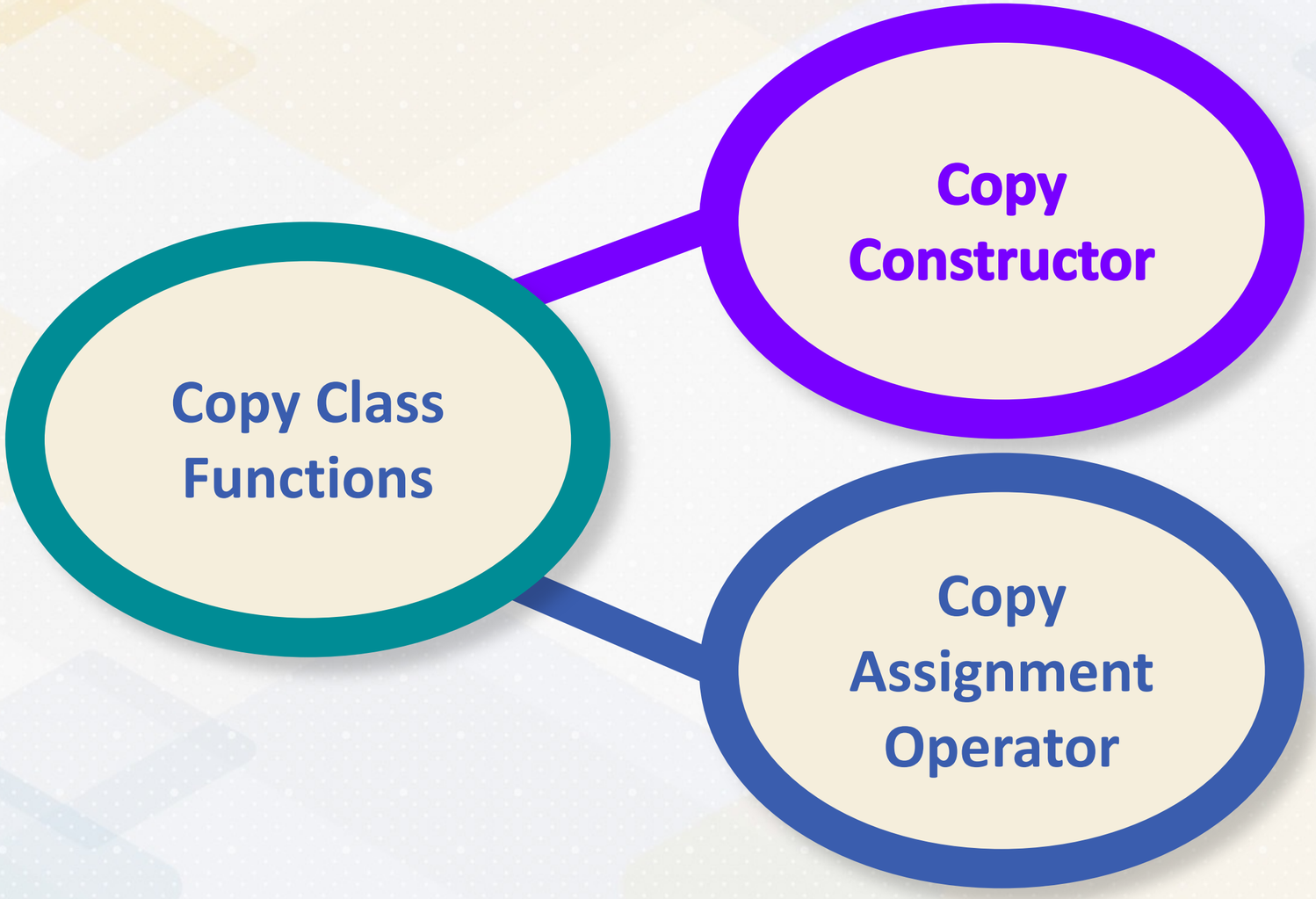
```
class Tuple {
    std::vector< std::unique_ptr < Field >> fields;


    public:
    void addField(std::unique_ptr< Field > field) {
        fields.push_back(std:: move(field))
    }
};
```

unique_ptr cannot be copied in the traditional sense to ensure unique ownership

Copy Semantics in C++



Copy Semantics in C++



Copy Constructor

```
Field(const Field &other) : type(other.type) {
  switch (type) {
  case STRING:
    data_s_length = other.data_s_length;
    data_s = std::make_unique<char[]>(data_s_length);
    strcpy(data_s.get(), other.data_s.get());
    break;
  }
}
```

Field field2 = field1;

Copy constructor creates new independent copy

Deep Copying

Copy Assignment Operator

A class method that assigns the contents of one Field object to another Field object: field1 = field2

```
Field &operator=(const Field &other) {
  if (this != &other) {
    switch (type) {
    case STRING:
      data_s_length = other.data_s_length;
      data_s = std::make_unique<char[]>(data_s_length);
      strcpy(data_s.get(), other.data_s.get());
      break;
    }
  }
  return *this;
}
```

Move Sematics in C++

While unique_ptr cannot be copied, it can be moved to transfer ownership of the managed object from one unique_ptr to another

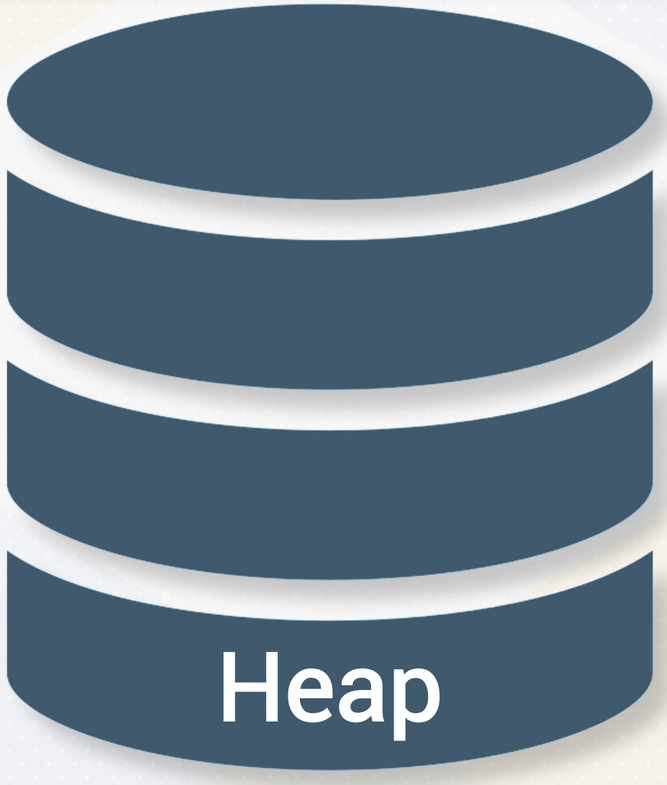
```
newTuple->addField(std::move(key_field))
```

HEAP vs STACK

Heap Allocation

Ideal for objects and data structures whose size cannot be determined at compile time

When you need the data to persist across different parts of the program

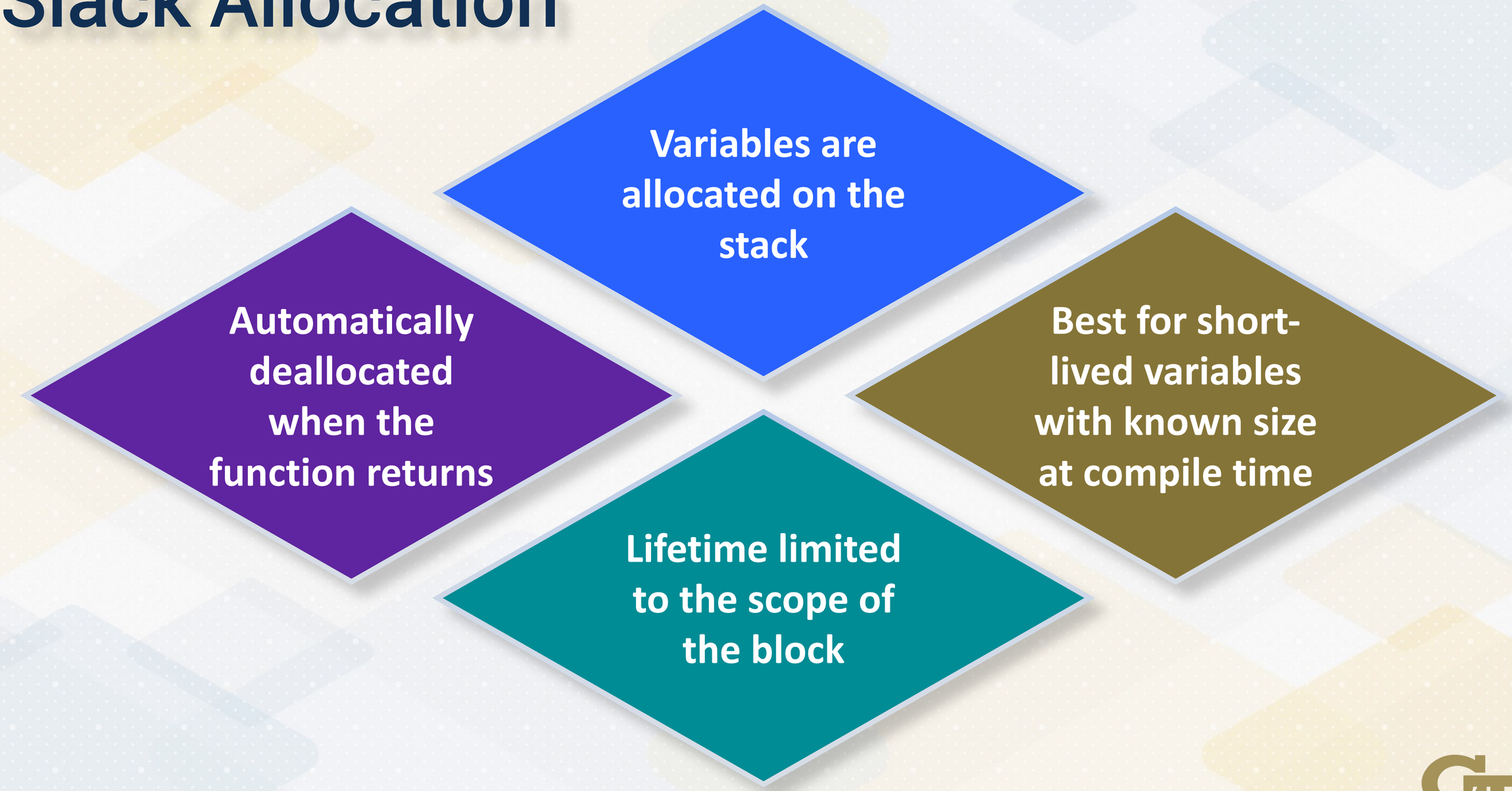


Heap-allocated variables can live beyond the scope in which they were created

Large pool of memory used for dynamic memory allocation

Allows flexible data management

Slack Allocation



Heap-Based Allocation

```
// Create and returns a smart pointer to a vector on the heap
std::unique_ptr<std::vector<int>> createVectorOnHeap() {
  return std::make_unique<std::vector<int>>(
      std::initializer_list<int>{5, 6, 7, 8});
}
int main() {
  auto heapVector = createVectorOnHeap();
  // heapVector now safely manages the lifetime of the vector
  // Safely accessing the vector
  for (int element : *heapVector) {
    std::cout << element << " ";
  }
  // The vector's memory is automatically cleaned up here
}
```

Slack-Based Allocation

```
// Attempts to return a reference to a vector allocated on the stack
const std::vector<int> & createVectorOnStack() {
    std::vector<int> vec = {1, 2, 3, 4};
    // Allocated on the stack
    return vec;
    // Warning: Returning reference to local/temporary object here
}
int main() {
    const auto & myVec = createVectorOnStack();
    // Undefined behavior: myVec refers to a destroyed vector
    // Attempting to use myVec will lead to undefined behavior
    for (int element: myVec) {std:: cout << element << " "; }
    // The program may crash, or worse, silently produce incorrect results
}
```

Heap vs Slack

Page Class

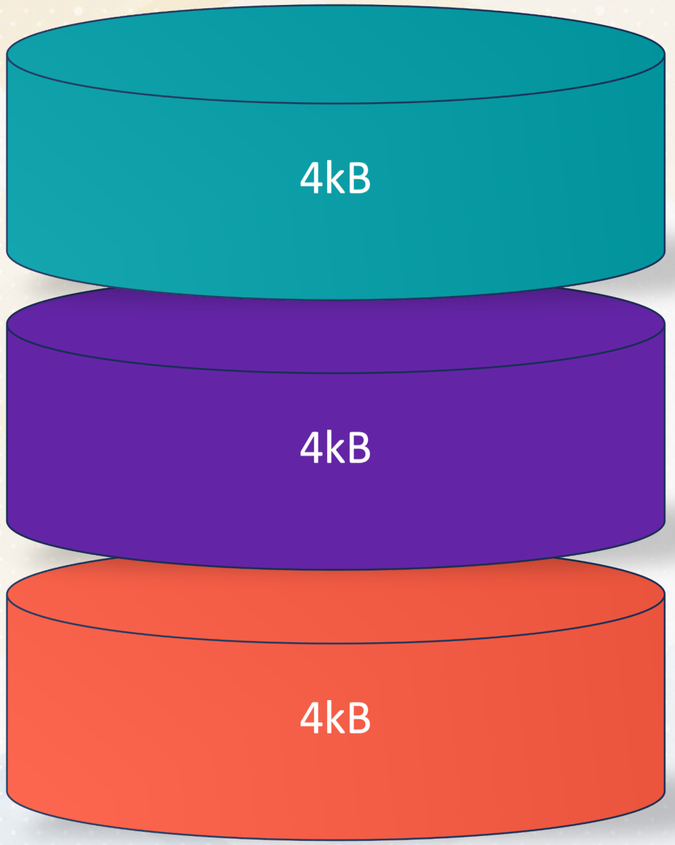
Persistence

Data Managed In-memory

Data Lost on Program Termination

Persistence Maintains Data Across Systems

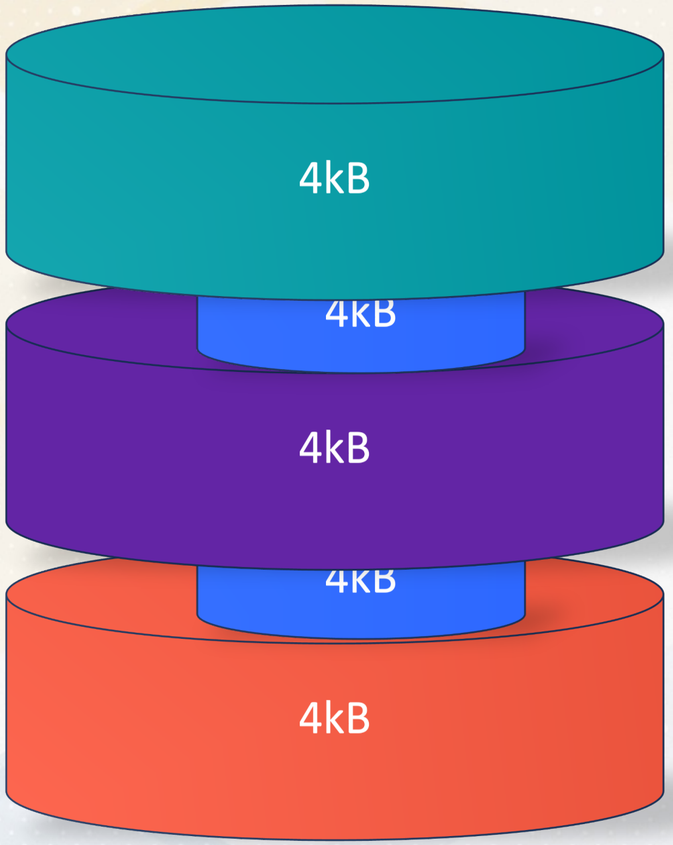
Disks and Pages



Basic Storage Units in Page Class

```
class Page {
public:
    size_t used_size = 0;
    std::vector<std::unique_ptr<Tuple>> tuples;
};
```

Page Class



Dynamic Tuple Management

Maintains Page Capacity

```
bool addTuple(std::unique_ptr<Tuple> tuple) {
    size_t tuple_size = // Calculated size;
    if (used_size + tuple_size > PAGE_SIZE) return false;
    tuples.push_back(std::move(tuple));
    used_size += tuple_size;
    return true;
}
```

Serialization

Serialization and Deserialization

Serialization

Deserialization

```
void write(const std::string& filename) const {
    // Logic to write page data to a file
}


void read(const std::string& filename) {
    // Logic to read page data from a file and reconstruct tuples
}
```

Serialization

Page Class Serialization Process

Write the number of tuples in Page

Loop through each tuple and serialize its data

```
void write(const std::string& filename) const {
        std::ofstream out(filename);
        // First write the number of tuples.
        size_t numTuples = tuples.size();
        out.write(reinterpret_cast<const
char*>(&numTuples), sizeof(numTuples));


}
```

Serialization

Page Class Serialization Process

Write the number of tuples in Page

Loop through each tuple and serialize its data

record how many fields each tuple contains

Serialization involves: *Field Type*, *Field Length*, *Field Data*

Serialization

```
// Then write each tuple.
for (const auto& tuple : tuples) {
  // Write the number of fields in the tuple.
  size_t numFields = tuple->fields.size();
  out.write(reinterpret_cast<const char*>(&numFields),
sizeof(numFields));


  // Then write each field.
  for (const auto& field : tuple->fields) {
          out.write(reinterpret_cast<const char*>(&field-
>type), sizeof(field->type));
          out.write(reinterpret_cast<const char*>(&field-
>data_length), sizeof(..));
          out.write(field->data.get(), field->data_length);
  }
}
```

Conclusion

- Smart Pointers
- Smart Field
- Heap vs Stack
- Page class
- Serialization