COMP 3522

Object Oriented Programming in C++

Week 12 day 1

Agenda

1. lvalue and rvalue
2. Move constructor & operator
3. Smart pointers

COMP 3522

lvalue AND rvalue

Introduction

"cannot bind **rvalue** reference of type 'int&&' to **lvalue** of type 'int'"

- While coding this term you've probably come across some compiler errors mentioning "lvalue" or "rvalue"
- Today we'll talk about what these lvalue and rvalues are, and finally understand these errors and handle them

C++ expressions

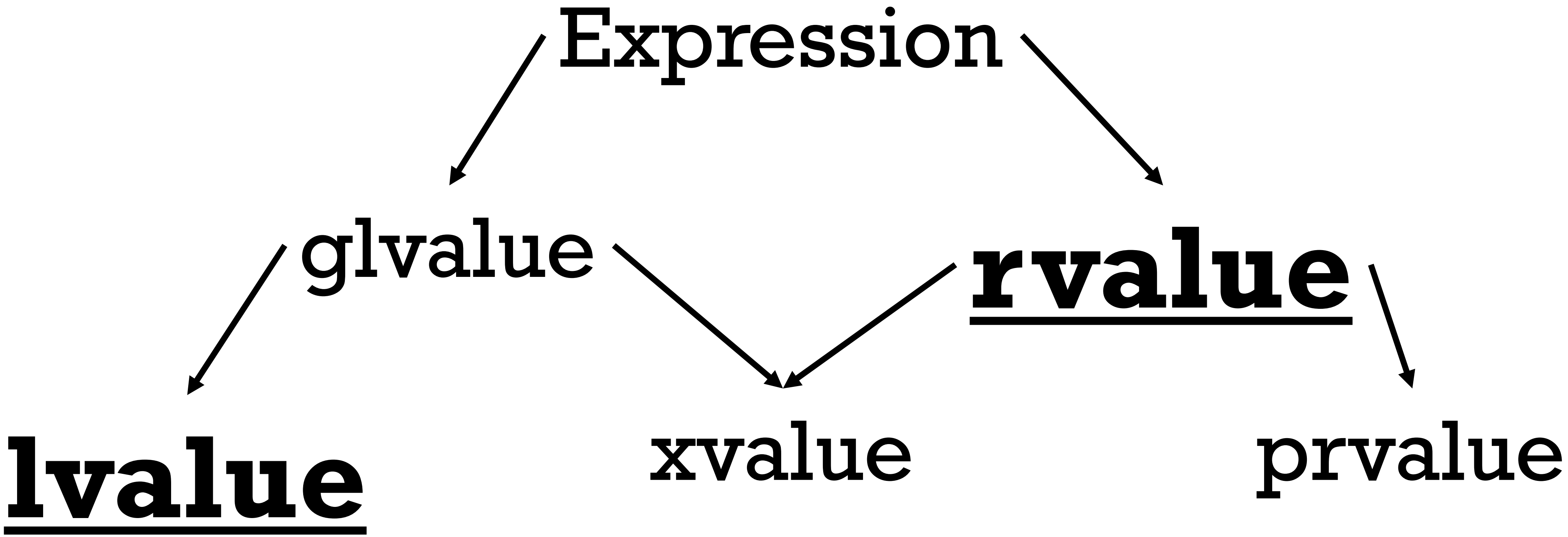
- Expression - sequence of operators and operands that specifies a computation
- Examples of expressions include:
  - An operator with its operands (x + y)
  - A literal (int, char, floating point, string, boolean, etc.) (5, ‘c’, 2.0, “hello”, true, etc)
  - A variable name or identifier, etc.
- **Order of evaluation** of arguments and subexpressions may generate intermediate results (int x = (a + b) + (c + d))
- Expressions have:
  1. **Type**
  2. **Category**

Value categories

- int x = 1;
- Every expression has a **type**
- Every expression belongs to a **value category**:
  1. glvalue
  2. prvalue
  3. xvalue
  4. **lvalue**
  5. **rvalue**
- Basis for rules the compiler uses for creating, copying, and moving temporary objects

https://en.cppreference.com/w/cpp/language/value_category

Value categories



int x = 1

lvalue

rvalue

Fun fact: Historically named because **lvalue** is **left** of the = symbol and **r**value is **right** of =.

```
int x = 1;
int y = 2;
int z = (x + y)
```

lvalue

lvalue

```
int x = 1;
int y = 2;
int z = (x + y)
```

Result of (x+y) is rvalue

lvalue

An object that persists beyond a single expression:

- has an **address**
- variables which have a **name**
- const **variables**
- array variables
- class **members**
- function calls which return an lvalue reference (&)

https://en.cppreference.com/w/cpp/language/value_category

rvalue

- Is not an lvalue
- **Temporary** value
- Has **no address** accessible by our program:
  - function call like std::move(x) which returns non-reference
  - increment and decrement
  - &a – result of memory address operation
  - function return types
    - int getNum() {}
    - int *getNumPtr() {}
  - literals
    - 42, true, nullptr
  - Comparison expression
    - a < b

https://en.cppreference.com/w/cpp/language/value_category

lvalue & rvalue function return

Functions that don't return by reference return rvalues

```
int function() {
    return 1;
}

cout << &function(); //ERROR!
```

lvalue & rvalue function return

Functions that return by reference return lvalues

```
int& function(int &num) {
    return num;
}

int x = 10;
cout << &function(x) << endl; //OK!
```

lvalue & rvalue exercise

```
//do these expressions work?
int i, j, *p;
i = 7;

7 = i;
j * 4 = 7;

*p = i;

const int ci = 7;
ci = 9;

((i < 3) ? i : j) = 7;
```

lvalue & rvalue exercise

```
//do these expression work?
int number = 10;

const int NAME_MAX = 20;

int* numberPtr = number;

std::map<string, double> scoreMap;

scoreMap["Lulu"] = 60.0;
```

lvalue & rvalue exercise

```
//do these expressions work?
int number = 10;

10 = number;
(number + 1) = 20;

int anotherNumber = 20;
int result = number + anotherNumber;

&number = 20;
```

lvalue & rvalue exercise

```
//what category of expression is the if?
int number1 = 10;
int number2 = 20;

if (number1 < number2)
{
// Do something
}
```

MOVE: CONSTRUCTOR AND OPERATOR

Pre-C++11: problem!

- Check out **problem.cpp**
- Pre-C++11 used to generate two copies!
- The first copy was made when the function returned a vector by value (copy constructor is implicitly invoked to generate the return value)
- The second copy was made when the return value was copied (again by the assignment copy) to scores
- How can we avoid making this extra copy?

Solution

1. rvalue reference
2. move semantics

What's an rvalue reference

- &&
- New operator introduced in C++11
- Functionally similar to reference operator &
- Operator **&** is for referencing an **lvalue**
- Operator **&&** is for referencing an **rvalue**

- Examine **`rvalue1.cpp`**

Consider std::move from <utility>

```
template< class T >
constexpr typename std::remove_reference<T>::type&&
move( T&& t ) noexcept;
```

Indicates that an object t may be "moved from"

aka converts an lvalue to an rvalue

aka forces "move semantics" on something even if it has a name

aka returns an rvalue that refers to the object passed as a parameter

aka **static casts to an rvalue reference type**

Move function does NOT actually move anything

It only converts an lvalue into an rvalue

What happens to the “original”

- Usually we don’t care because it’s a temporary anyway
- Accessing it yields an **unspecified value**
- **Should only be destroyed or assigned a new value**

- Check out **move.cpp**

myvector.push_back (foo);

Vector myVector

foo-string

string foo

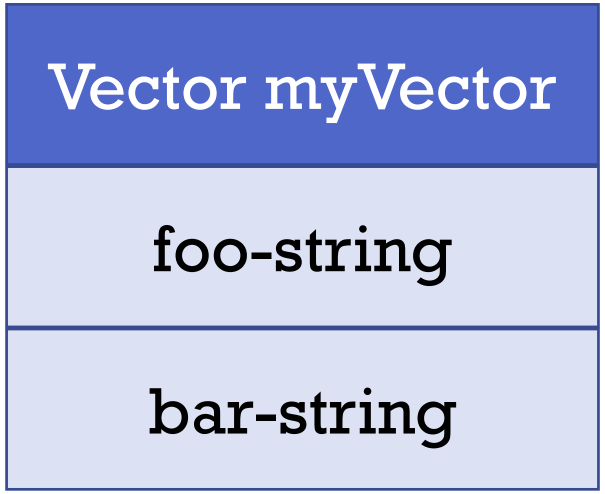
foo-string

“foo-string” copied from foo and pushed into myVector

string bar

bar-string

myvector.push_back (std::move(bar));



- "bar-string" moved from bar and pushed into myVector
- Vector's overloaded push_back accepts rvalue parameter. "Moves" that data into vector

Another constructor…

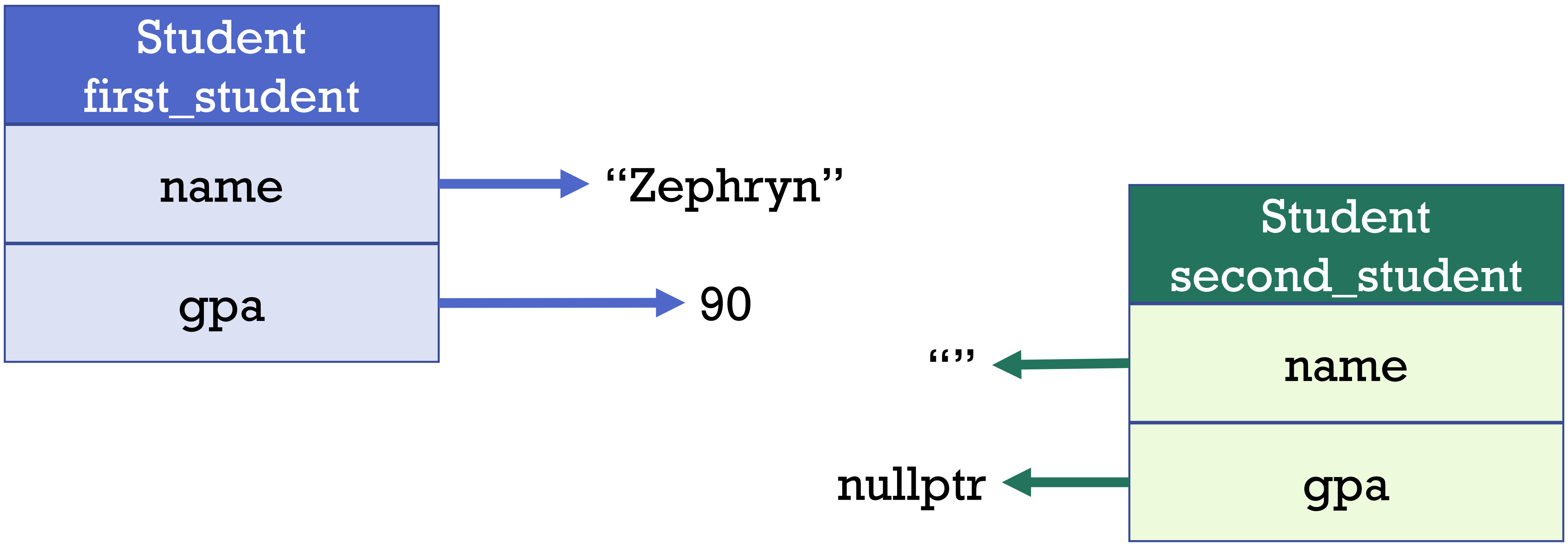
- We need some way to manage this new move semantic
- Introducing move assignment and the move constructor
- Recall standard member functions:
    1. Default constructor C()
    2. Copy constructor C(const C&)
    3. Copy assignment C& operator=(const C&)
    4. Destructor ~C()
    5. **Move constructor**
    6. **Move assignment**

Move constructor

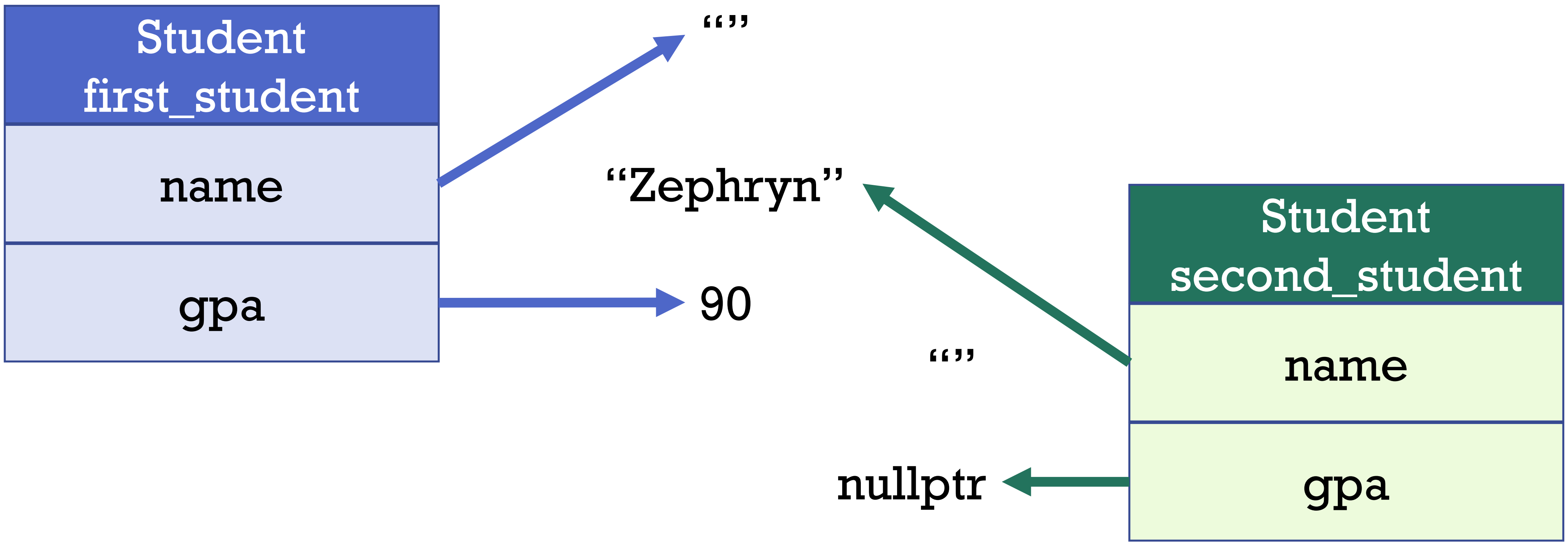
```
ClassName:: ClassName(ClassName&& other)
{
	//...
}
```

- Takes ownership of member variables from another object
- Faster, avoids memory allocation (unlike copy constructor)
- Kind of "shallow copy"
- Check out **Simplemove.cpp**, **move2.cpp**, **move3.cpp**

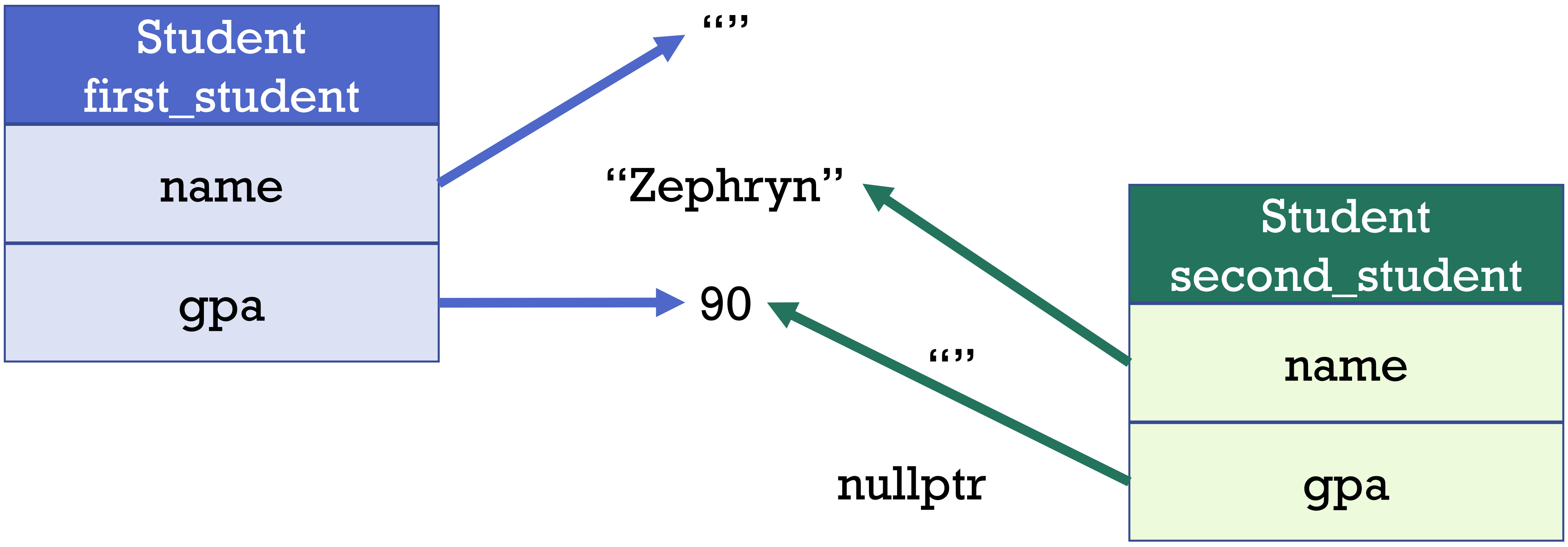
```
student(student&& other) : name{move(other.name)}
{
        gpa = other.gpa;
        other.gpa = nullptr;
}
student second_student(std::move(first_student));
```



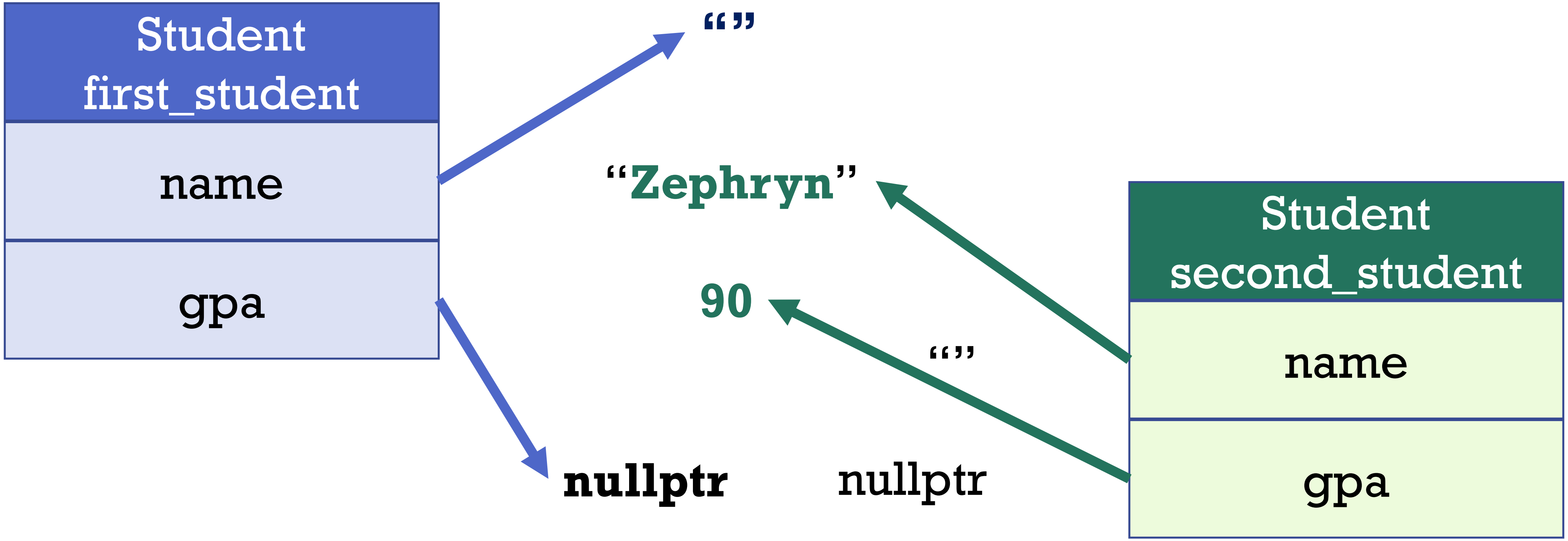
```
student(student&& other) : name{move(other.name)}
{
        gpa = other.gpa;
        other.gpa = nullptr;
}
student second_student(std::move(first_student));
```



```
student(student&& other) : name{move(other.name)}
{
        gpa = other.gpa;
        other.gpa = nullptr;
}
student second_student(std::move(first_student));
```



```
student(student&& other) : name{move(other.name)}
{
        gpa = other.gpa;
        other.gpa = nullptr;
}
student second_student(std::move(first_student));
```



Move assignment operator

```
ClassName& ClassName::operator=(ClassName&& other)
{
    //...
}
```

- Same concept as move constructor
- Acquires ownership of member variables
- Avoids memory reallocation (fast!)
- Shallow copy
- Examine **move4.cpp**

Another look...

```
//function that creates and returns MyClass rvalue
instance
MyClass createMyClass(){
  return MyClass{};
};

MyClass foo; // default constructor
MyClass bar = foo; // copy constructor
MyClass baz = createMyClass(); // move constructor
foo = bar; // copy assignment
baz = createMyClass(); // move assignment
```

But why, tho?

- **The concept of moving is most useful for objects that manage the storage they use**
- Consider objects that allocate storage with **new** and **delete**
- In such objects, copying and moving are really different operations:
- **Copying** from A to B means that new memory is allocated to B and then the entire content of A is copied to this new memory allocated for B
- **Moving** from A to B means that the memory already allocated to A is transferred to B without allocating any new storage. It involves simply copying the pointer. MORE EFFICENT! No new dynamic memory allocated

ACTIVITY

1. Modify your matrix class from A1, so that it has a move constructor and a move assignment operator.
2. Test that they work by using your matrix and the move constructor and move assignment in a main method. Prove they work by printing the contents of your matrices before and after their use.

UNIQUE, SHARED, AND WEAK POINTERS

The problem with dynamic allocation

- We **dynamically allocate memory** for objects and data objects on the heap/free store using **new**
- We must **remember to deallocate** the memory using **delete** before losing the pointer
- The **alternative is a memory leak**

What’s a memory leak?

- When a pointer to a patch of dynamically allocated memory goes out of scope before the memory is returned to the free store
- That memory is now unavailable to the running code
- It is inaccessible
- As our application runs, it will exhaust available memory!

- C++ doesn’t have a Java-style garbage collector to take care of this for us

Example

- Think back to RPN lab

- **rpn_calculator::operation_type** dynamically allocates an operation object and returns a pointer to it
- **rpn_calculate::perform** accepts the pointer and uses the operation object to do some math

- Did you remember to delete the pointer?
- If not, that's a memory leak!

Okay, well then let's just use static allocation

- Static allocation: **memory allocated at compile time in the stack** or other data elements
- Local variables are deleted/destroyed automatically from stack memory when we exit a function
- No pointers!
- No memory management!

- Can't have memory persist "outside" the function it's created in
  - Restricts how to design systems if can't create memory within function

C++11 introduced a smart solution

Smart Pointers!

1. **unique_ptr**
2. **shared_ptr**
3. **weak_ptr**

Smart pointer

- #include <memory>
- A class object that **acts like a pointer** but has additional features
- **Encapsulates** a ‘raw’ pointer
- Helps us manage dynamic memory allocation
- **When the smart pointer goes out of scope, its destructor uses delete to free the memory it encapsulates.**

1. unique_ptr

- Template
- Wraps a ‘raw’ pointer
- Ensures the pointer it contains is deleted on destruction (like a garbage collector!)
- Automatically deletes the object it encapsulates using a stored deleter when:
  - Destroyed (goes out of scope)
  - Value changes by assignment
  - Value changes by call to reset function
- Let’s examine **unique_ptr.cpp**

When do we use a unique_ptr?

1. We can replace the use of pointers for data members in classes (see **unique_use_1.cpp**)
2. Use for local variables inside functions (see **unique_use_2.cpp**)
3. Use inside STL collections (next slide for details)

STL containers

- Value semantics = lots of copies = lots of overhead
- So let's use a pointer, right?
- Before we delete the container, we have to delete the contents
- What if we forget? MEMORY LEAK!

- Try unique pointers! (see **unique_use_3.cpp**)

2. shared_ptr

- A unique_ptr is unique, it cannot be shared or copied

- What if we want aliases?
- How do we make sure the memory is not destroyed until all the aliases are out of scope?

- We use a shared_ptr

shared_ptr

- Uses **reference counting**
- Keeps a count of how many shared_ptr objects are holding the same pointer (which we can view using the **use_count** function)
- Reference counting **uses atomic functions and is thread-safe**
- Each shared_ptr releases co-ownership when it goes out of scope:
  - Destroyed (goes out of scope)
  - Value changes by assignment or a call to reset function

shared_ptr

- When all shared_ptrs are out of scope, the memory is deleted (limited garbage collection, again!)
- shared_ptr objects can only share ownership by copying their value
- If two shared_ptr are constructed from the same raw pointer, they will both consider themselves the sole owner
- This can cause potential access problems when one of them deletes its managed object and leaves the other pointing to an invalid location
- Check out **shared_ptr_1.cpp, shared_ptr_2.cpp**

An important fact about make_shared

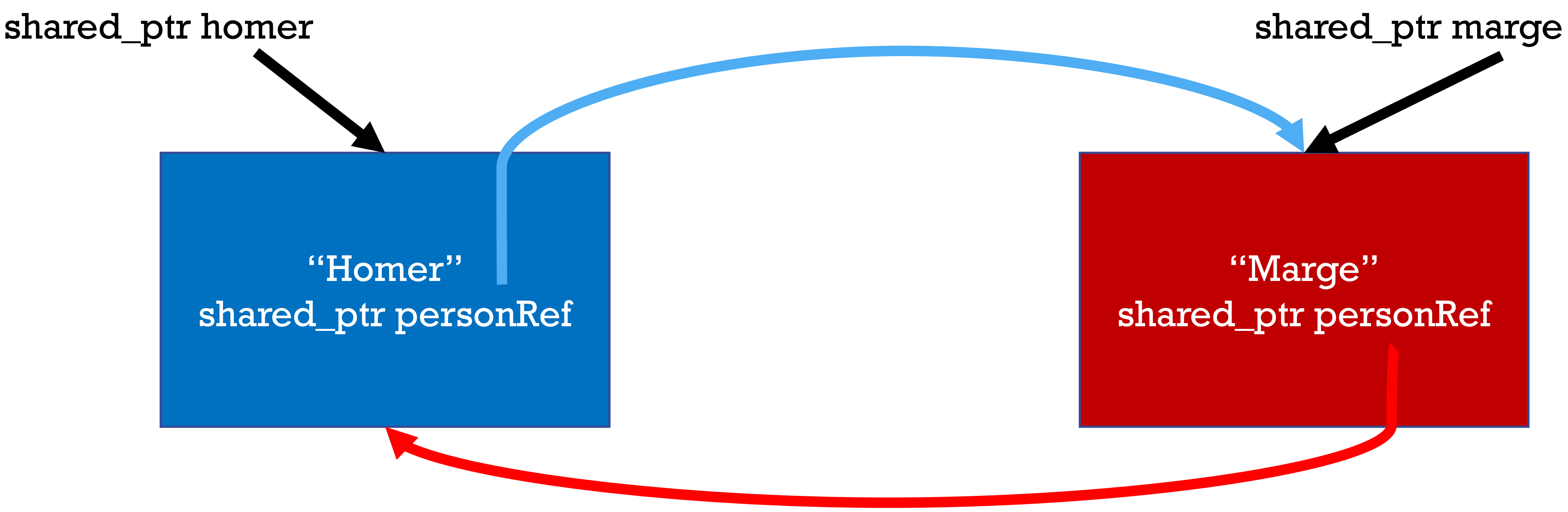
Though it is possible to create a shared_ptr by passing a pointer to its constructor, **constructing a shared_ptr with make_shared should be always preferred**.

It is **more efficient** (only requiring one memory allocation rather than two).

3. weak_ptr

- Holds a non-owning reference to a pointer managed by shared_ptr
- Must be converted to a shared_ptr to access the object
- Models temporary ownership
- Check out weak_ptr_1.cpp, **weak_ptr_2.cpp**, weak_ptr_3.cpp, weak_ptr_4.cpp

3. weak_ptr circular reference



- Primarily used in rare cases to break circular references, i.e., in doubly linked lists
- Homer and Marge's personRef shared pointer has reference count 2
- Can't call destructor on shared pointers if reference count > 1

weak_ptr_circle.cpp

Smart pointer guidelines

- When an object is dynamically allocated, immediately assign it to a smart pointer that will act as its 'owner'
- If a program will need more than one pointer to an object, use shared_ptr
- If a program doesn't need multiple pointers to the same object, use a unique_ptr

- Final word: check out **code_snippet_1.cpp**

Final things

- Send Anonymous topic requests about anything we've covered in the course
- https://forms.gle/3DJvQB1WraGzeB7p7
- Student survey time