COMP 3522

Object Oriented Programming in C++

Week 11 day 2

Agenda

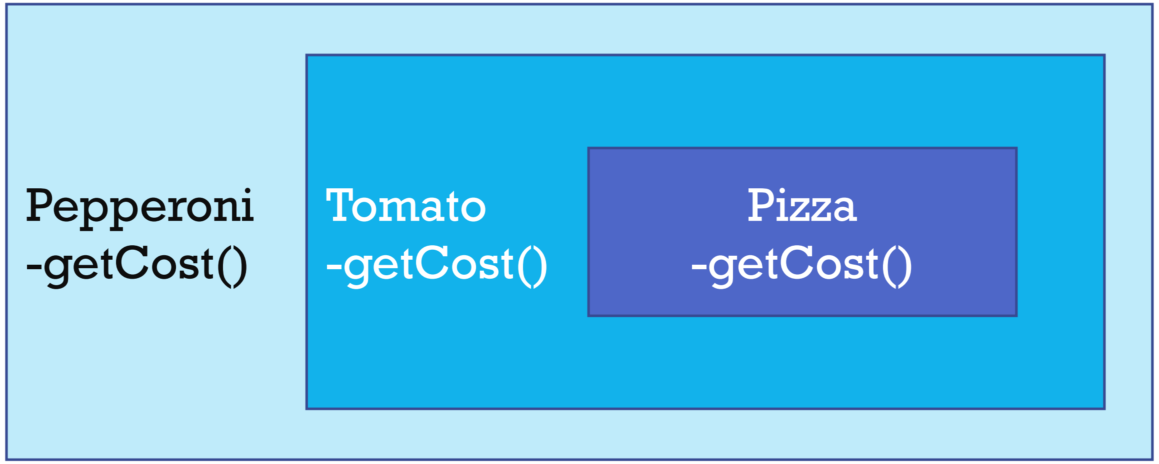
1. Design patterns
    1. Decorator
    2. Facade
    3. Proxy
    4. Strategy
2. lvalue and rvalue
3. Move constructor & operator
4. Smart pointers

COMP 3522

DECORATOR

Introduction - Decorator

- We'll "decorate" our pizza object with toppings
  - Wrap our pizza object with Tomato topping
  - Wrap that with another topping, Pepperoni

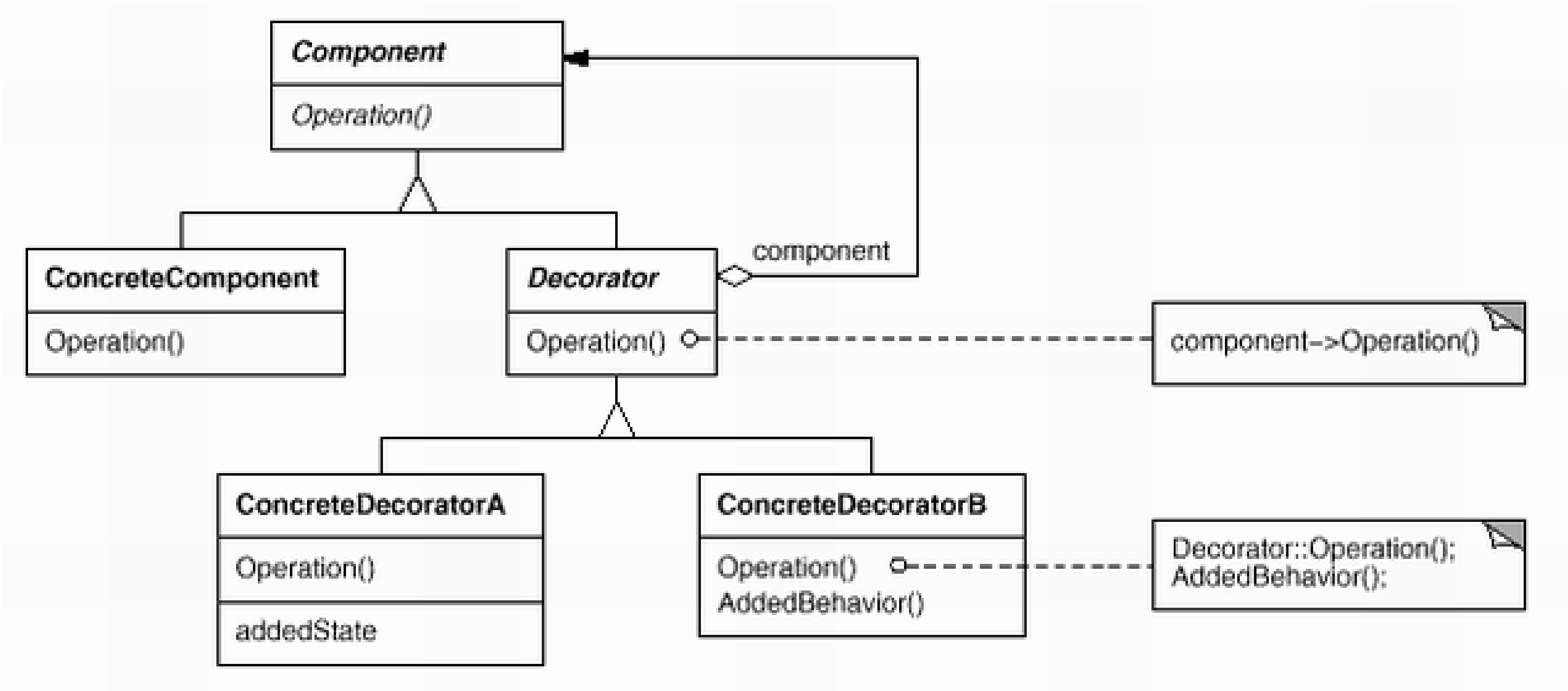


Summary

- We want to attach additional responsibilities to an object dynamically, not to an entire class
- We don’t want a huge pile of subclasses
- So we wrap it
- We want to enclose the component in another object that adds the additional responsibility
- The enclosing object is called a decorator
- The decorator conforms to the interface of the component it decorates so that its presence is transparent to clients

Description

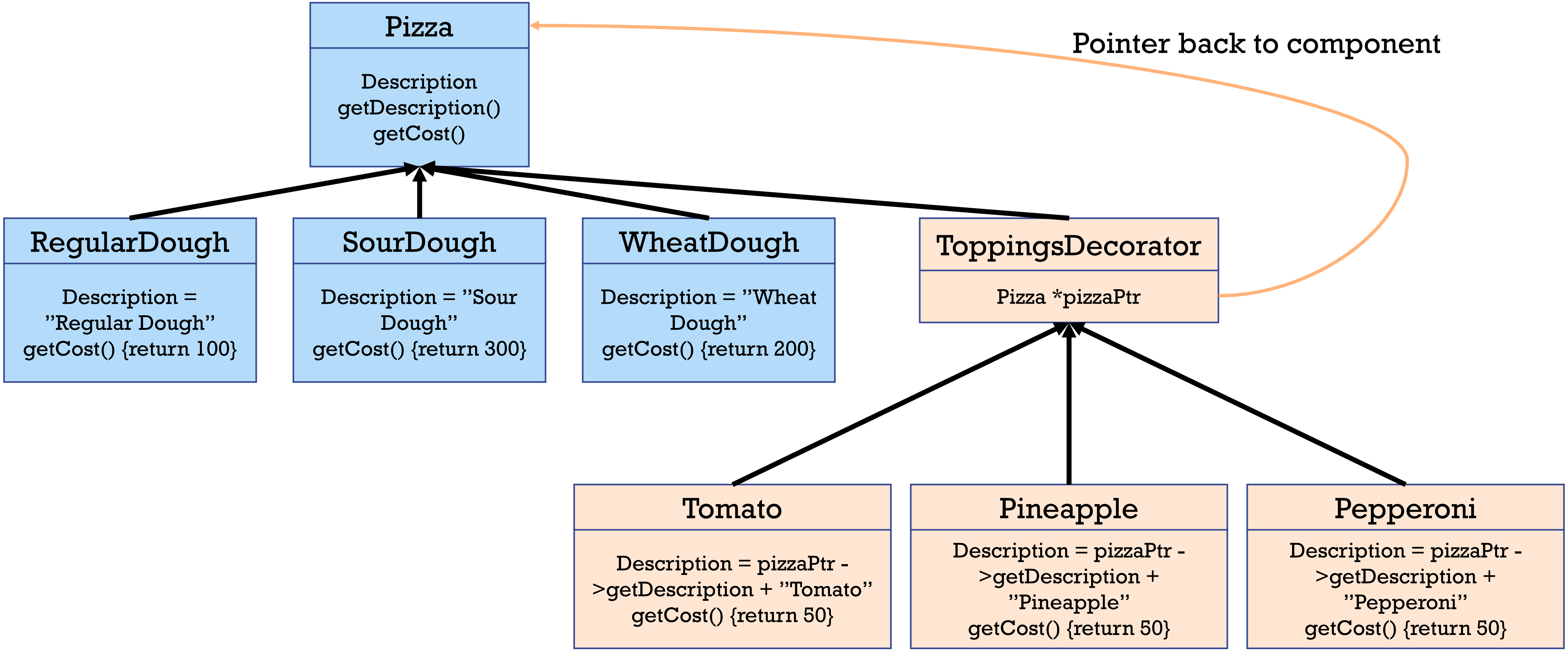
- The decorator forwards requests to the inner component and may perform additional actions before or after forwarding



Parts

- Component defines the interface for objects that have responsibilities added to them dynamically
- ConcreteComponent defines an object to which additional responsibilities can be attached
- Decorator maintains a reference to a Component object and defines an interface that conforms to Component's interface
- ConcreteDecorator adds responsibilities to the Component

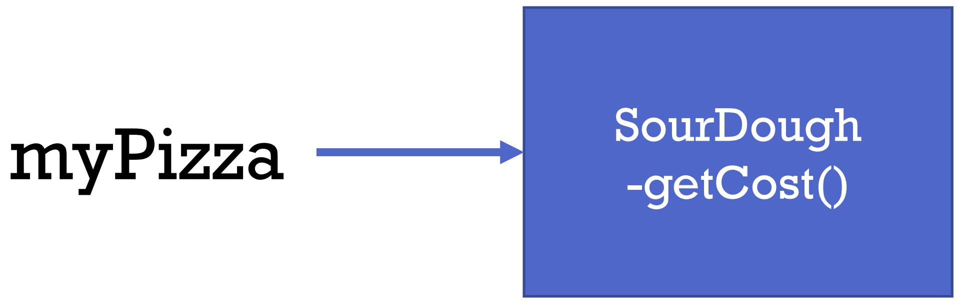
Pizzas…



Pizza
Description
getDescription()
getCost()
Pointer back to component
RegularDough
Description = "Regular Dough"
getCost() {return 100}
SourDough
Description = "Sour Dough"
getCost() {return 300}
WheatDough
Description = "Wheat Dough"
getCost() {return 200}
ToppingsDecorator
Pizza *pizzaPtr
Tomato
Description = pizzaPtr ->getDescription + "Tomato"
getCost() {return 50}
Pineapple
Description = pizzaPtr ->getDescription + "Pineapple"
getCost() {return 50}
Pepperoni
Description = pizzaPtr ->getDescription + "Pepperoni"
getCost() {return 50}

Introduction - Decorator

- Pizza *myPizza = new SourDough{};



- Create SourDough pizza base and have pointer point to it

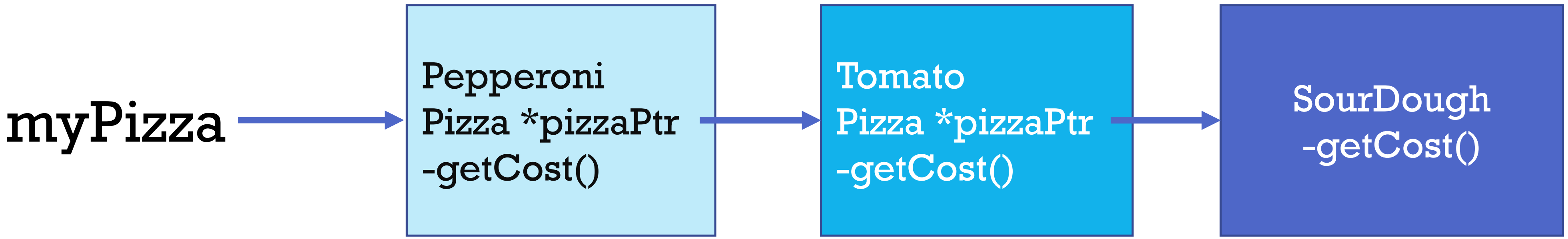
Introduction - Decorator

- Pizza *myPizza = new SourDough{};
- myPizza = new Tomato(myPizza);

- Create Tomato object, set its internal pizzaPtr to point to myPizza
- Tomato's grandparent base class is Pizza, so the pointer it returns from dynamic memory can be saved into pizza pointer

Introduction - Decorator

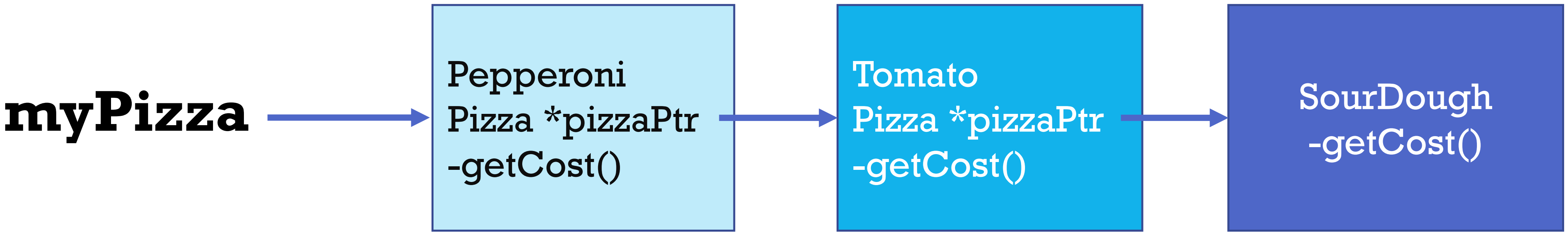
- Pizza *myPizza = new SourDough{};
- myPizza = new Tomato(myPizza);
- myPizza = new Pepperoni(myPizza);



- Create Pepperoni object, set its internal pizzaPtr to point to base Pizza
- Pepperoni's grandparent base class is Pizza, so the pointer it returns from dynamic memory can be saved into pizza pointer

Introduction - Decorator

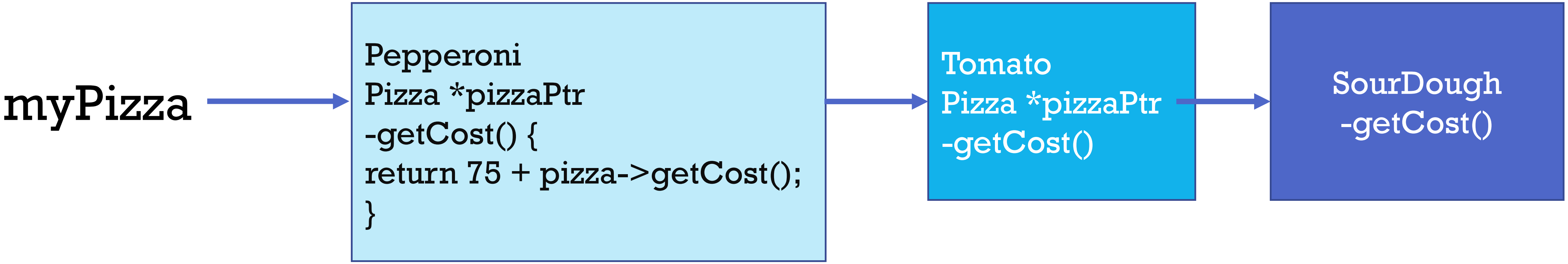
- Pizza *myPizza = new SourDough{};
- myPizza = new Tomato(myPizza);
- myPizza = new Pepperoni(myPizza);
- myPizza->getCost();



- Notice how starting from **myPizza pointer**, it looks like a linked list
- Get cost of total pizza by calling myPizza->getCost();

Introduction - Decorator

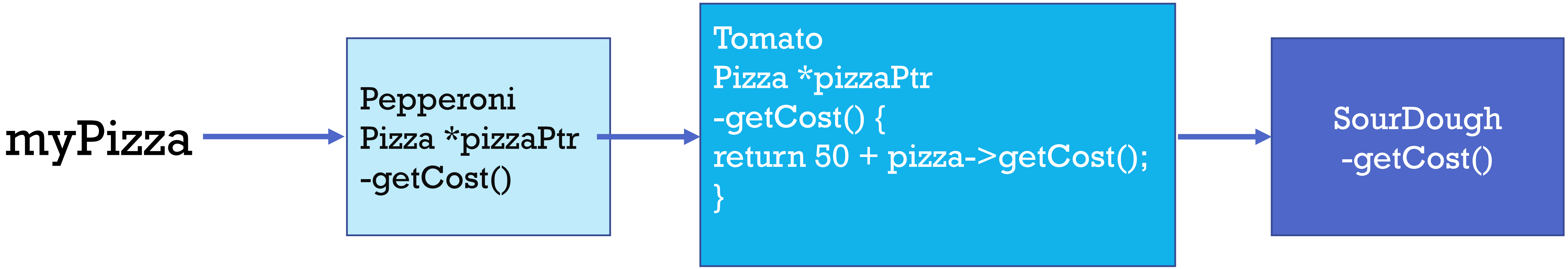
- Pizza *myPizza = new SourDough{};
- myPizza = new Tomato(myPizza);
- myPizza = new Pepperoni(myPizza);
- myPizza->getCost();



- Calls will chain into toppings until they reach SourDough

Introduction - Decorator

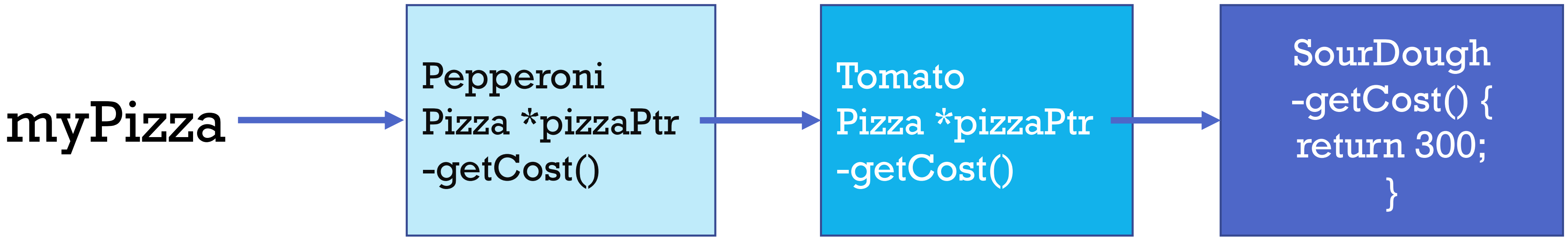
- Pizza *myPizza = new SourDough{};
- myPizza = new Tomato(myPizza);
- myPizza = new Pepperoni(myPizza);
- myPizza->getCost();



- Calls will chain into toppings until they reach SourDough

Introduction - Decorator

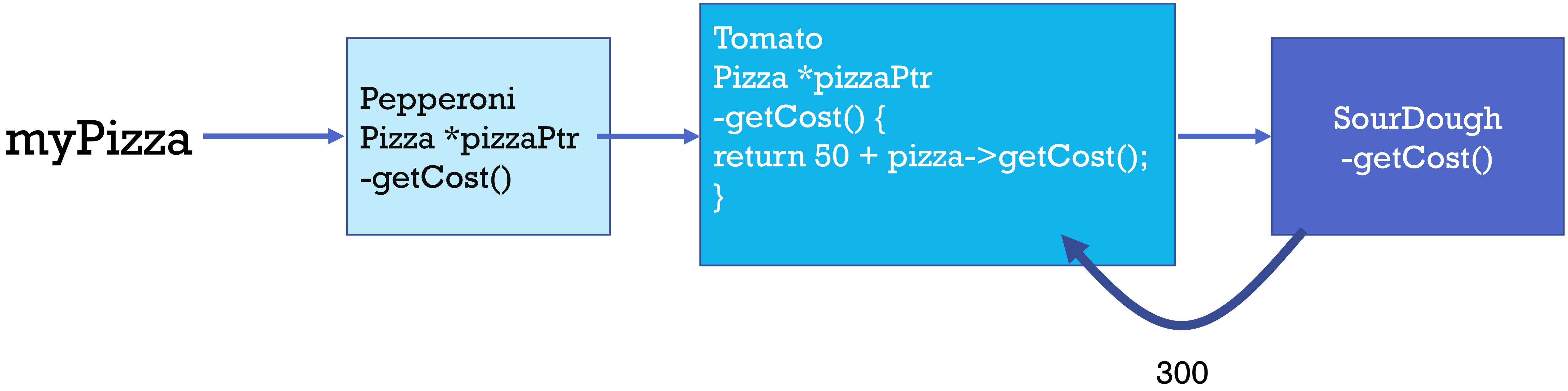
- Pizza *myPizza = new SourDough{};
- myPizza = new Tomato(myPizza);
- myPizza = new Pepperoni(myPizza);
- myPizza->getCost();



- Costs added up as we return from getCost() function calls

Introduction - Decorator

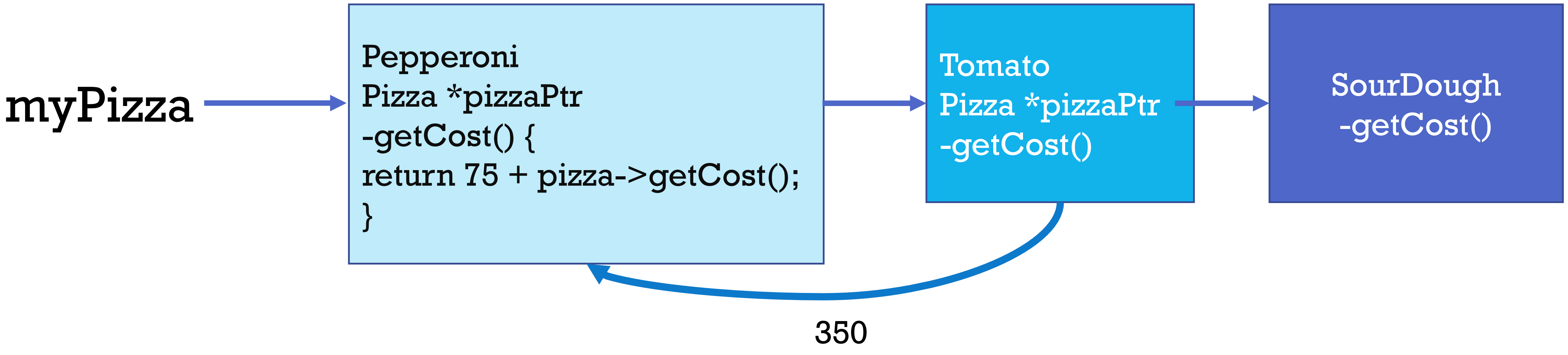
- Pizza *myPizza = new SourDough{};
- myPizza = new Tomato(myPizza);
- myPizza = new Pepperoni(myPizza);
- myPizza->getCost();



- Costs added up as we return from getCost() function calls

Introduction - Decorator

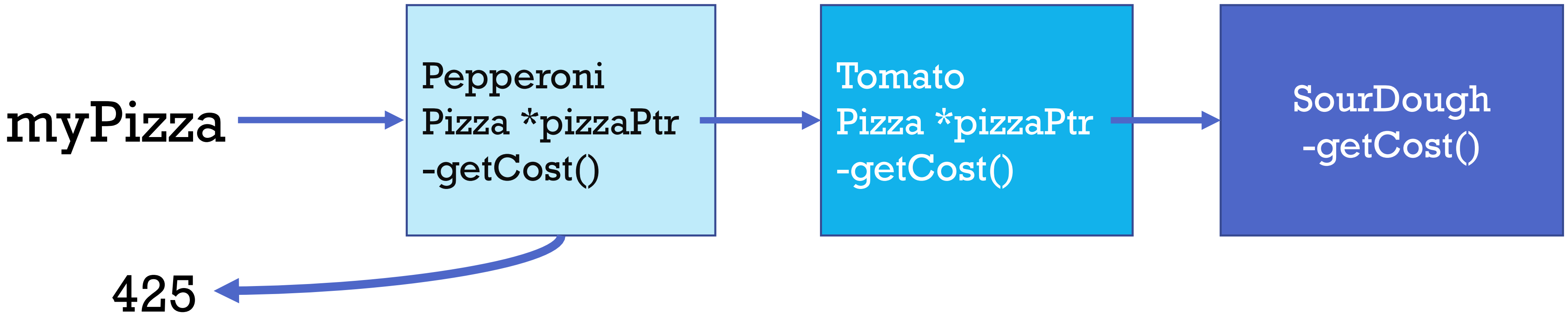
- Pizza *myPizza = new SourDough{};
- myPizza = new Tomato(myPizza);
- myPizza = new Pepperoni(myPizza);
- myPizza->getCost();



- Costs added up as we return from getCost() function calls

Introduction - Decorator

- Pizza *myPizza = new SourDough{};
- myPizza = new Tomato(myPizza);
- myPizza = new Pepperoni(myPizza);
- myPizza->getCost();



- Costs added up as we return from getCost() function calls

Benefits

- More flexible than static inheritance
  - Responsibilities can be added and removed at runtime simply by attaching and removing them
- Avoids feature-laden classes high up in the hierarchy
- Works best when we keep the Component classes lightweight:
  - Focus on defining an interface, not storing data
  - Defer the definition of data storage to subclasses.

Decorator.cpp

FAÇADE

Introduction

- Imagine you just created the the ultimate home theatre system
- In order to watch a movie, however, you must:

1. Dim the lights
2. Pull down the screen
3. Turn on the projector
4. Set the projector input to BlueRay Input
5. Turn on the amplifier
6. Set the amplifier input to BlueRay Input
7. Set the volume
8. Turn on the BlueRay player
9. Put the disc into the BlueRay player…

Motivation

- That’s too complicated
- We need a simple interface for the home theatre system:
  - It can let us perform common tasks easily
  - We still have full access to the whole system if we need to make changes
- We want to provide a single interface to communicate with a set of interfaces in a subsystem
- We define a **Façade**, a higher level interface to wrap the others

Description

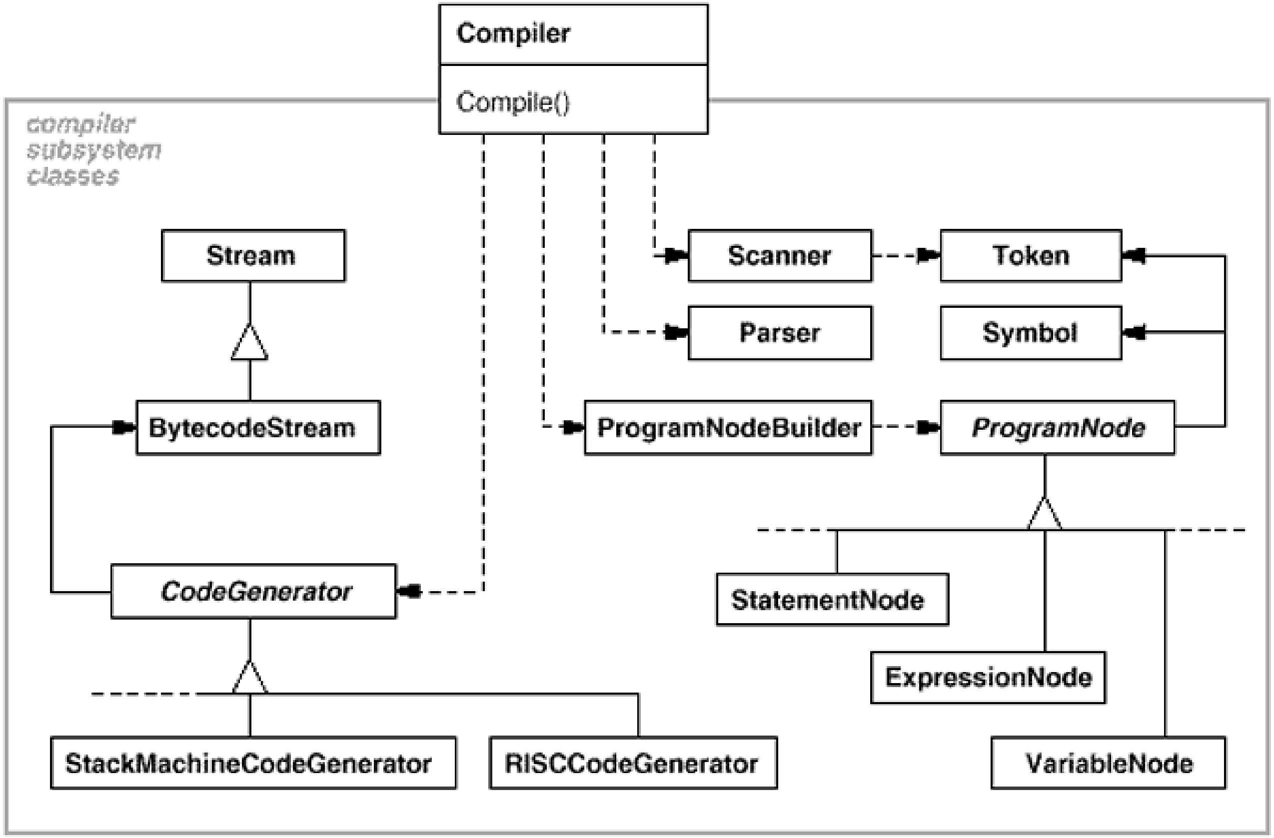
- Façade design pattern hides complexity of a system and provides an interface to the client through which they can access the system
- Clients communicate with the subsystem by sending requests to the Façade
- The Façade forwards the request to the correct subsystem recipient
- The Façade may need to translate its interface to the subsystem interfaces
- Clients never have direct access to the subsystem
- Usually only one Façade is required, so it can also be a Singleton!

Real world example

- Imagine organizing a wedding for 100 people
- To organize this event you must:
  - Find and decorate hall
  - Find a band
  - Buy flowers
  - Send invitations
  - Figure out food, etc
- Or…you can hire an event manager. The event manager handles all the organization for you.
- The event manager is a façade

Canonical example

- A Compiler is actually a front for:
    1. Scanner
    2. Parser
    3. Program nodes
    4. Program node builders
    5. All sorts of streams and generators, etc…
- We know and need to know nothing about these complex and powerful submodules
- The compiler is a façade!



When should we use a façade…

- Use façade when there are many dependencies between clients and the implementation classes of an abstraction
- The façade decouples the subsystem from the clients and other subsystems to promote
  - Independence
  - Portability
- Promotes loose coupling
- Shields clients from subsystem components, reducing the number of objects that a client has to deal with

Façade.cpp

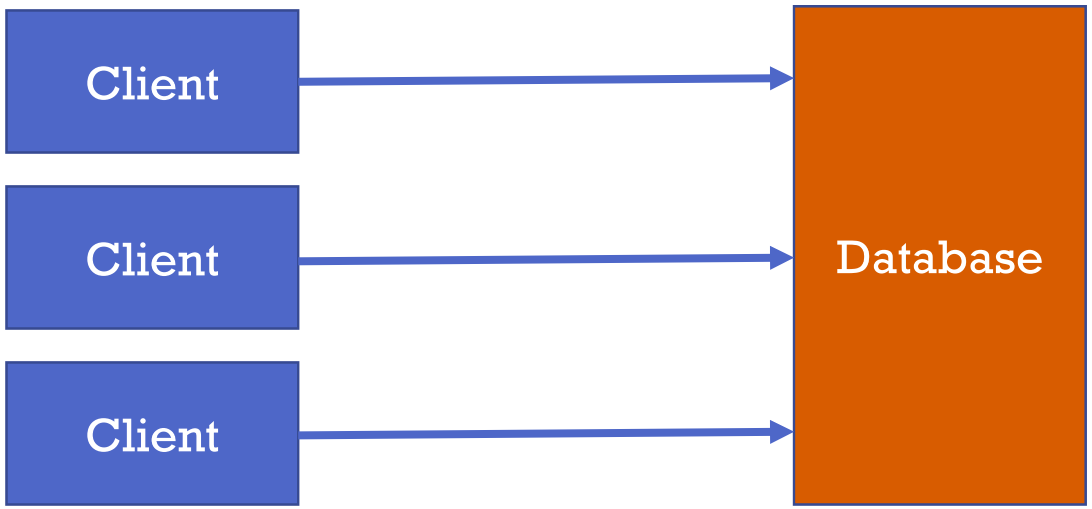
PROXY

Proxy

- Structural design pattern
- Allows us to substitute the original object for a "proxy" object
- The proxy gives access to the original object, allows operations to be performed before or after requests occur on original object

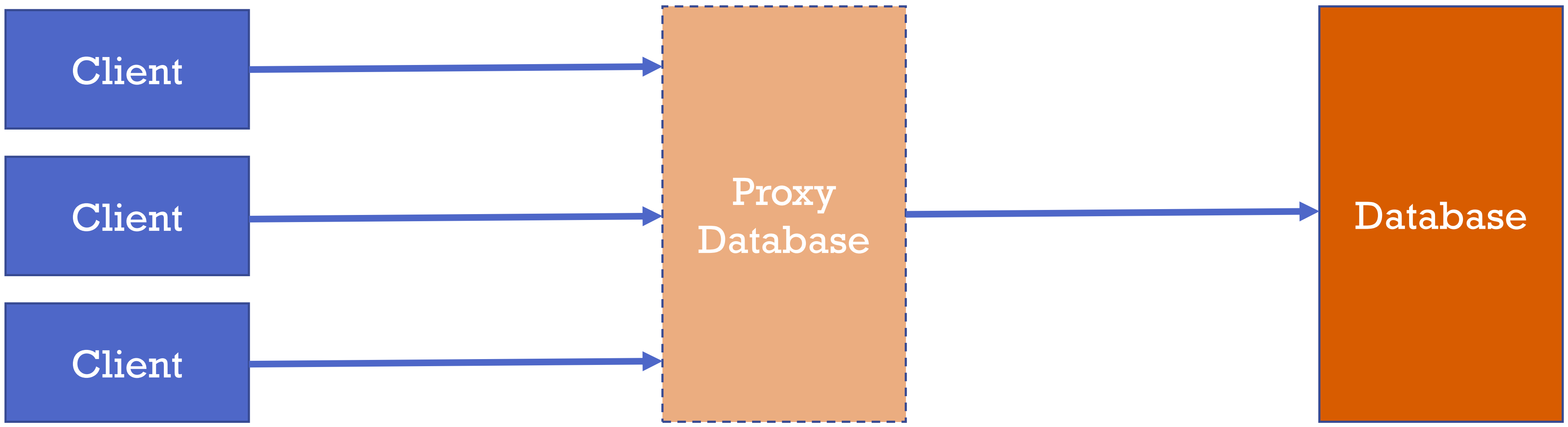
Proxy

- Imagine you have a database
- Multiple clients want to access that database
- Accessing the database is slow



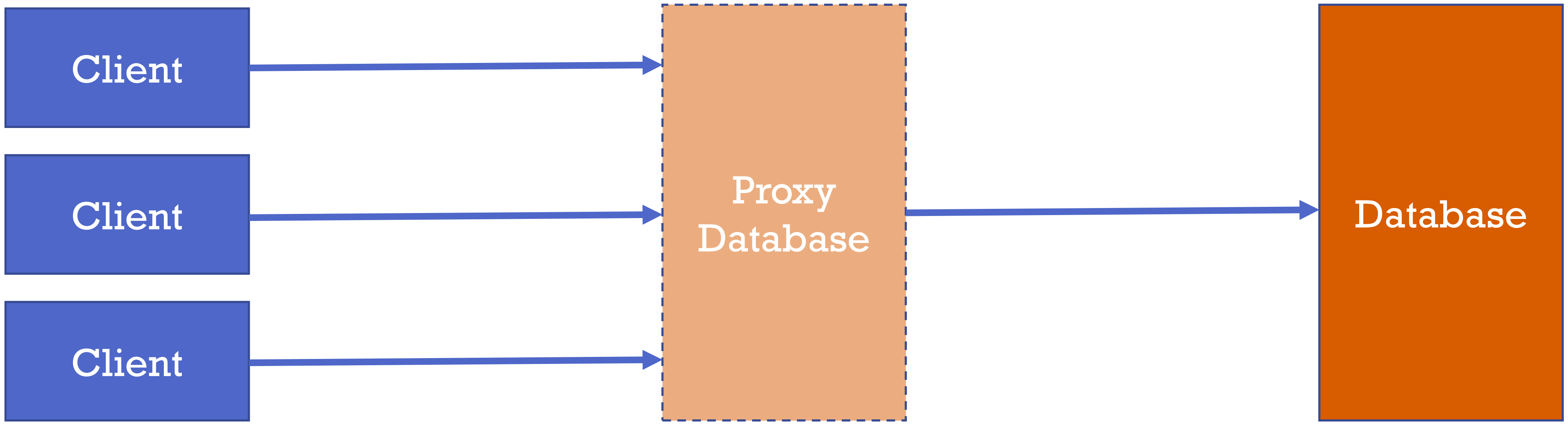
Proxy

- Instead of clients getting direct access to client, they get access to a proxy version of the database
- From the client's perspective, they don't realize they're accessing a proxy



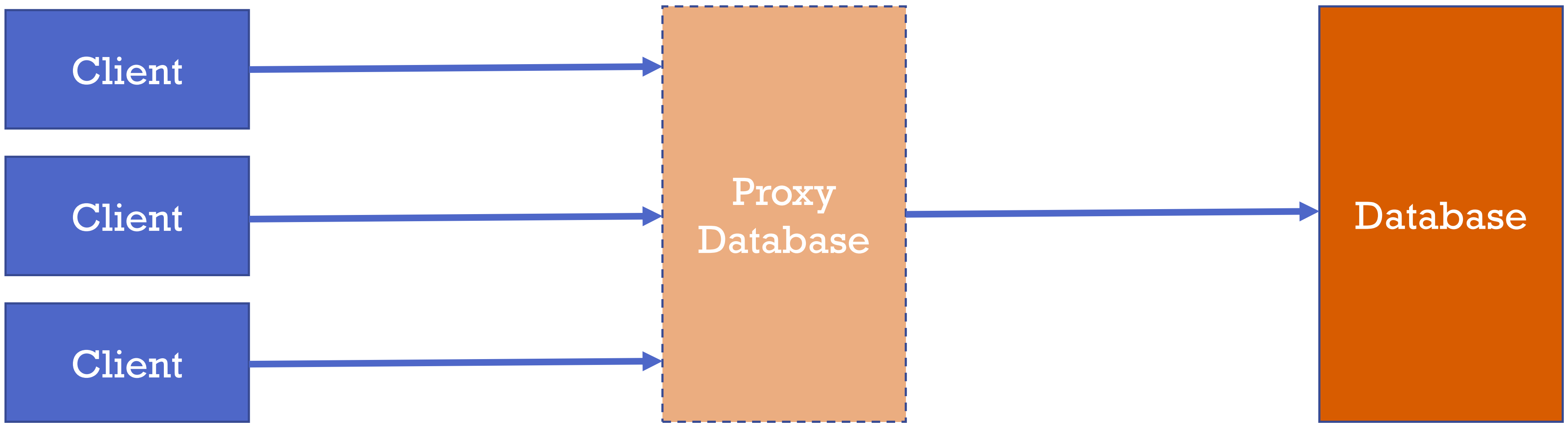
Proxy

- To create the illusion that the clients are interacting with the original database, the proxy database must implement the same interface as the real database



Proxy

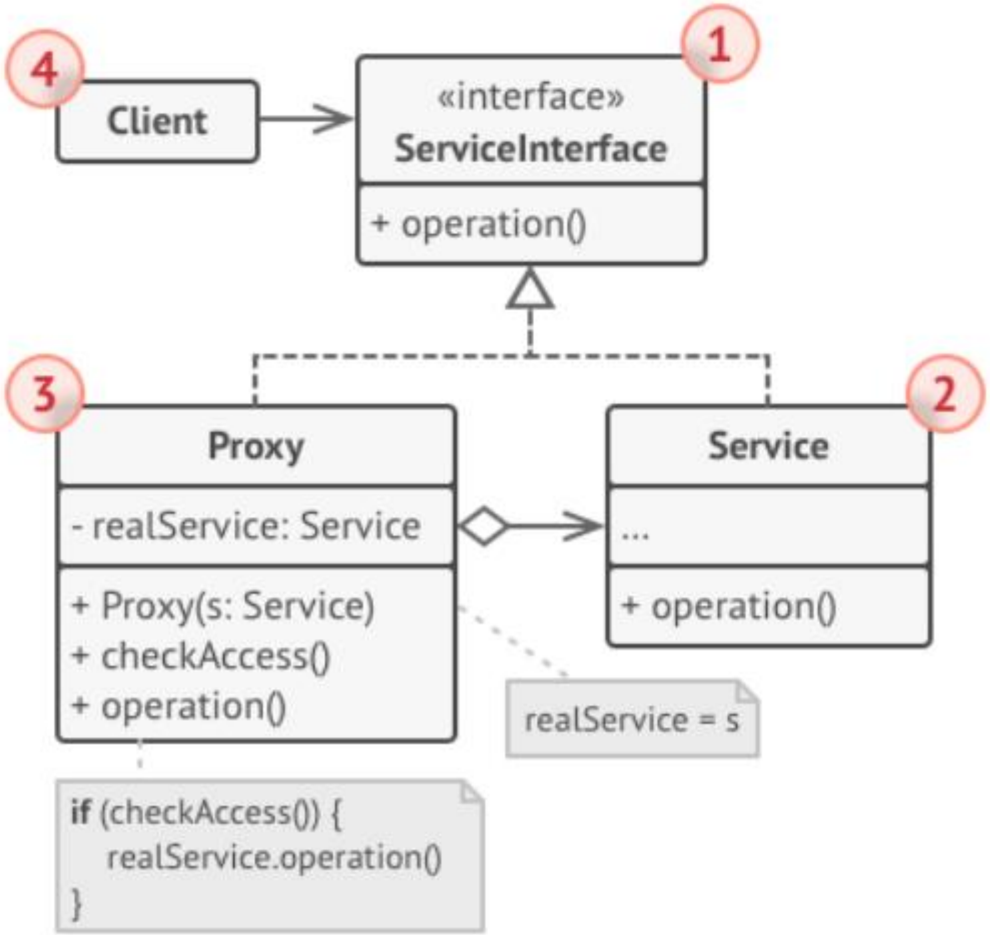
- Main benefit: Can execute code before or after calls to the original database
- This allows us to add additional functionality without modifying the database class



Real world example

- Think of money
- Money is represented as dollar bills/coins
- Not safe to keep large amounts of dollar bills at home
- So we use a Bank as a proxy for our money
  - Bank adds additional features for our money
  - Security, interest etc
- Credit cards are an additional proxy to our banks
  - Adds even more features
  - Purchase items before we have to pay
  - Cashback, travel points etc

Proxy class diagram



https://refactoring.guru/design-patterns/proxy

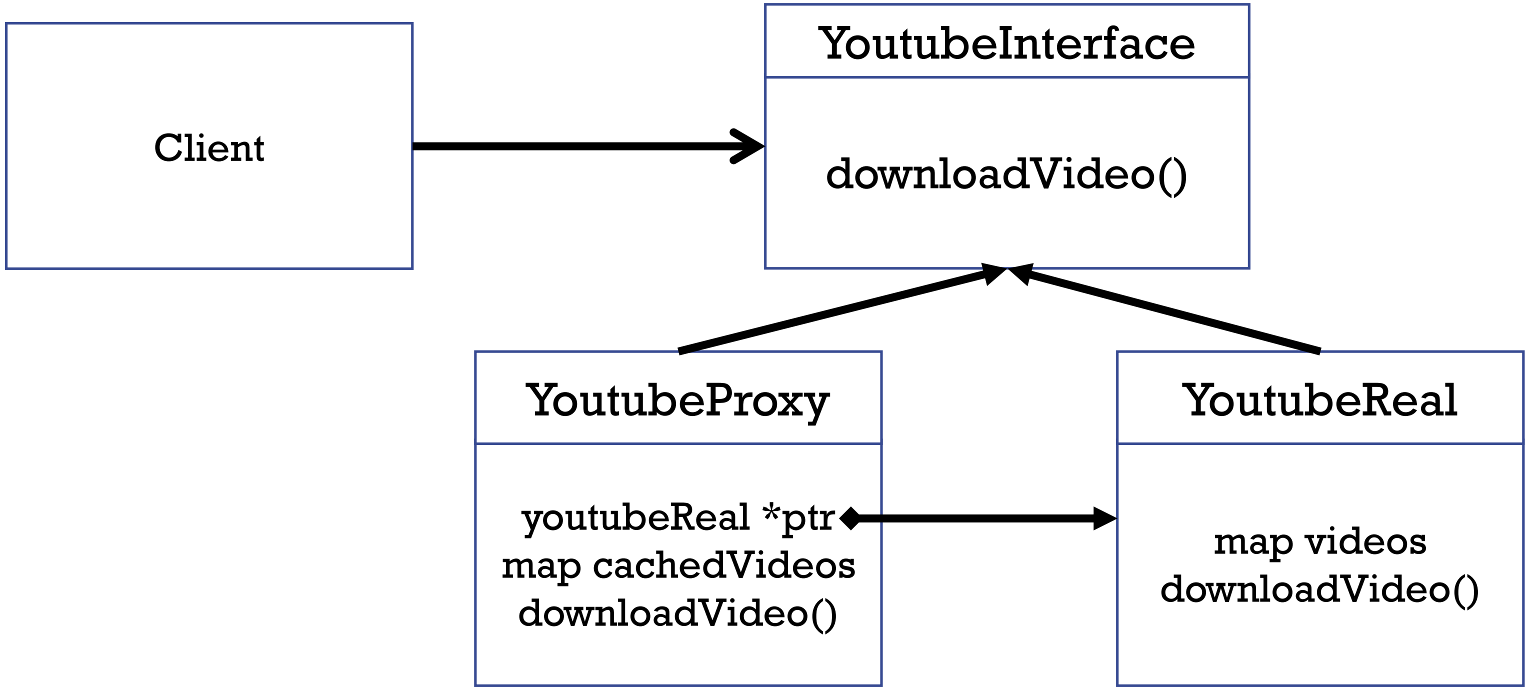
Proxy example

- Let's create a Youtube video downloading proxy
- Motivation: Having one server to handle all video requests is slow
  - If one user watches the same video multiple times, we should cache the video instead of downloading every time from youtube server

Proxy example

- Goal: Design a system where from the client's perspective, they're accessing youtube directly. In reality they're accessing a youtube proxy
- YoutubeProxy
  - Accesses the real youtube if a new video is requested
  - Return a cached version if a repeat video is requested
- YoutubeReal
  - Returns actual video given a video url
- YoutubeInterface
  - Provides interface that both YoutubeProxy and Real must implement

Proxy example



proxy.cpp

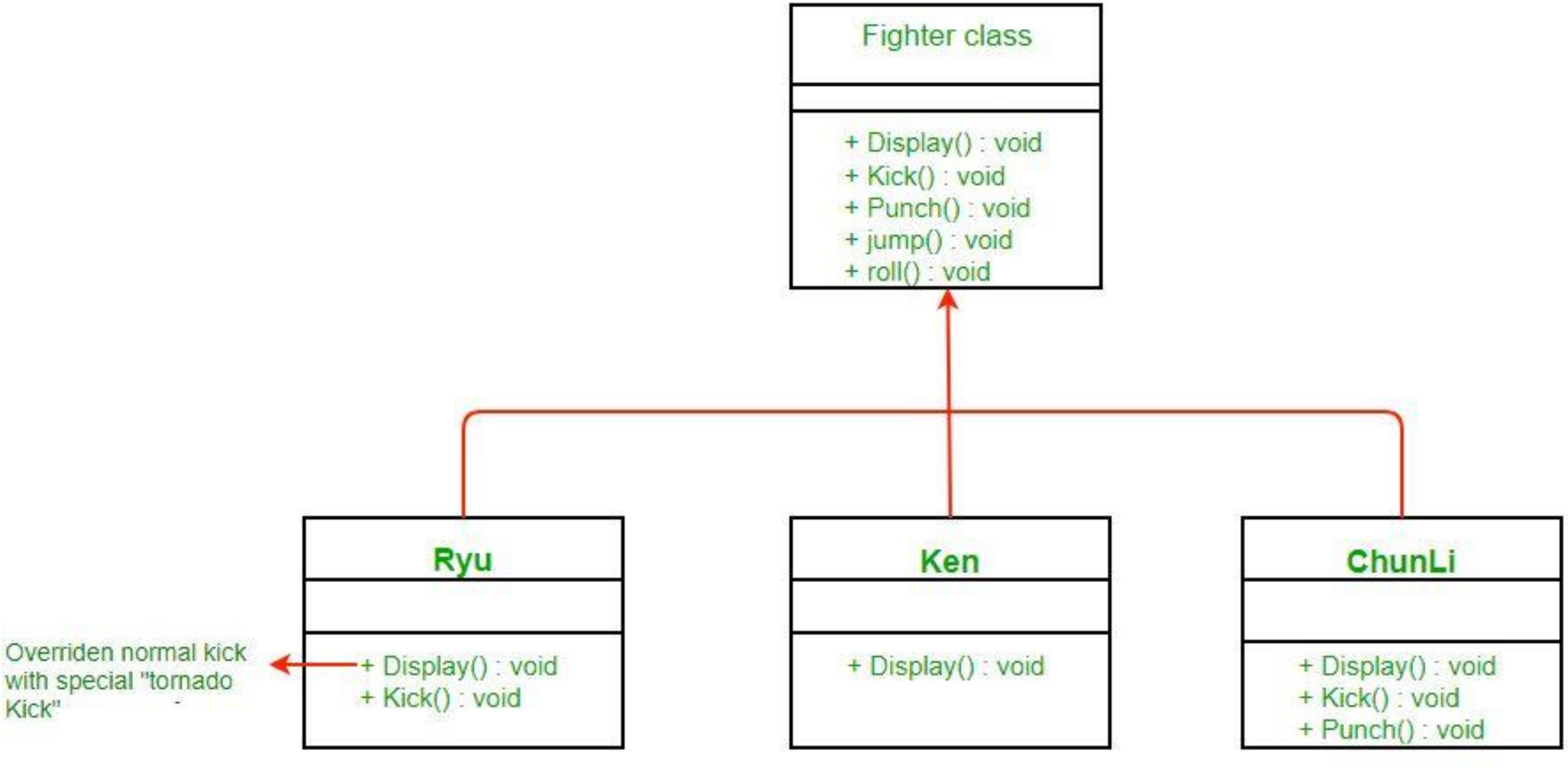
STRATEGY

Introduction

- Imagine you're creating a 2 player fighting game like "Street Fighter"
- In this fighting game, all fighters can perform four possible moves
  - Kick, punch, roll, and jump
- We model the fighter with by placing common features into a Fighter class, and subclass specific fighters

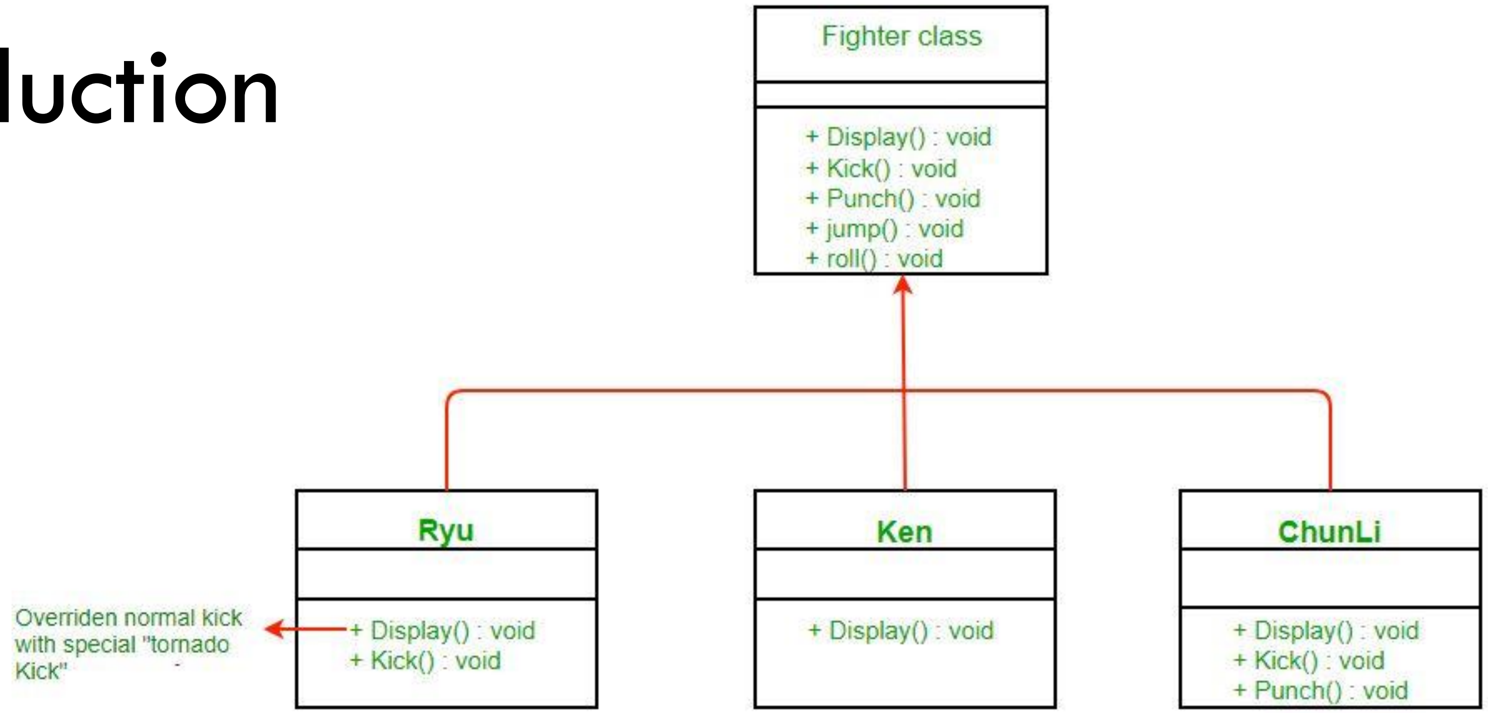
https://www.geeksforgeeks.org/strategy-pattern-set-1/

Introduction



https://media.geeksforgeeks.org/wp-content/uploads/classinh.jpg

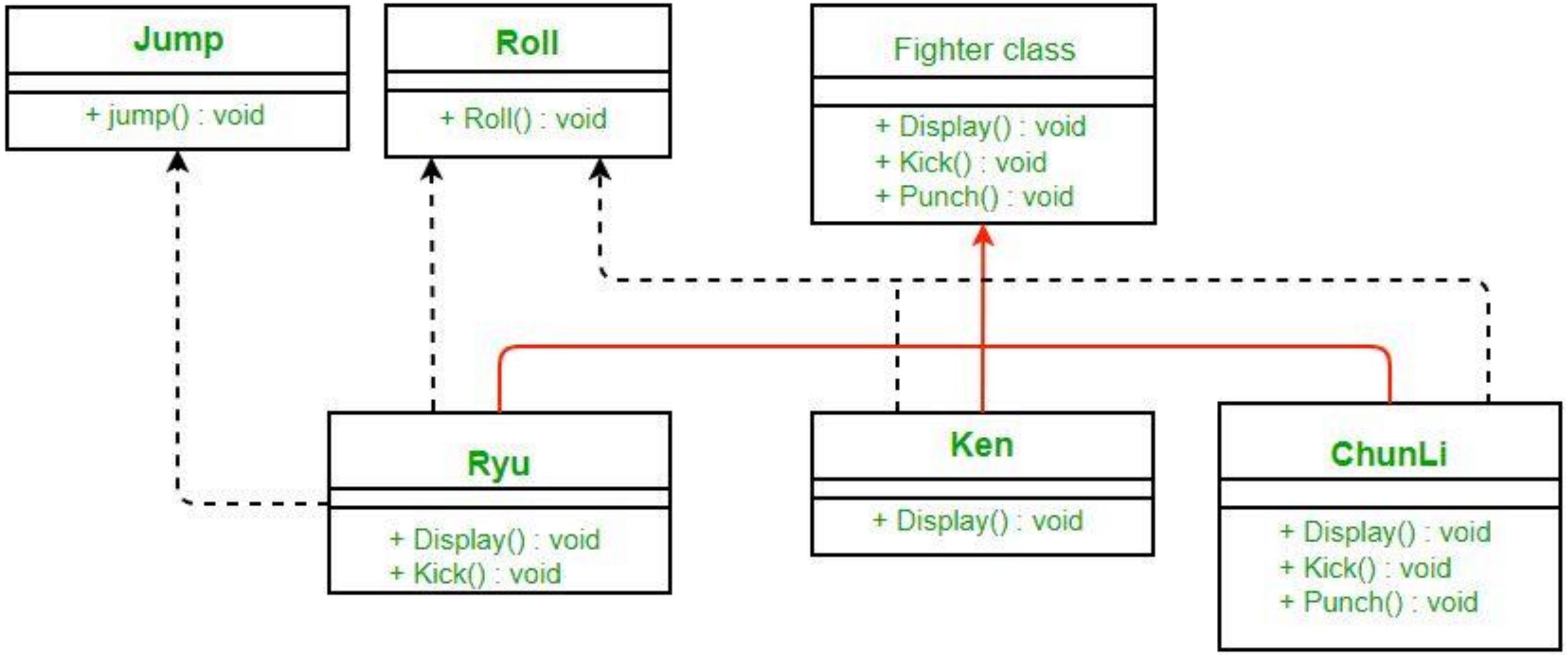
Introduction



- There are some design problems with this approach
- What if a character can not jump?
- Still inherits jump function from superclass
- Can override jump function to do nothing
    - But will have to worry about other classes that don’t jump
    - Worry about future sub-classes that don’t jump

https://media.geeksforgeeks.org/wp-content/uploads/classas.jpg

Introduction

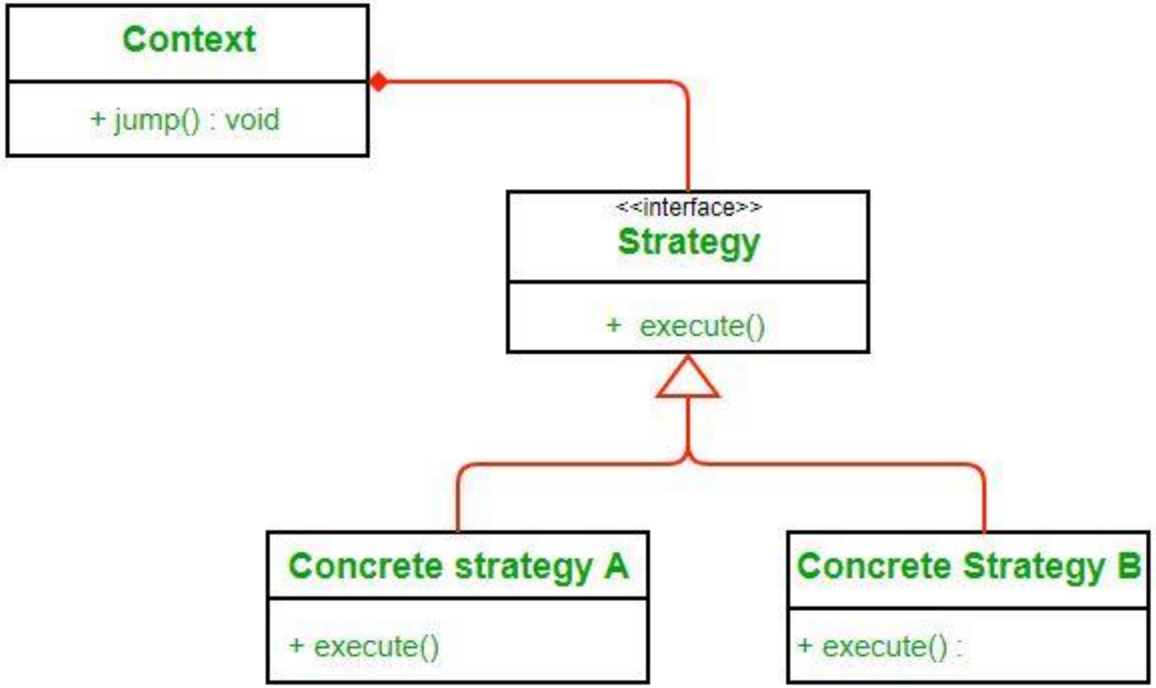


- Can try to abstract out optional moves into interfaces
- Only subclasses that use those moves implement the interfaces
- Problem with this approach is code duplication
  - Jump/Roll code may be duplicated for every character that implements the interface

Strategy pattern

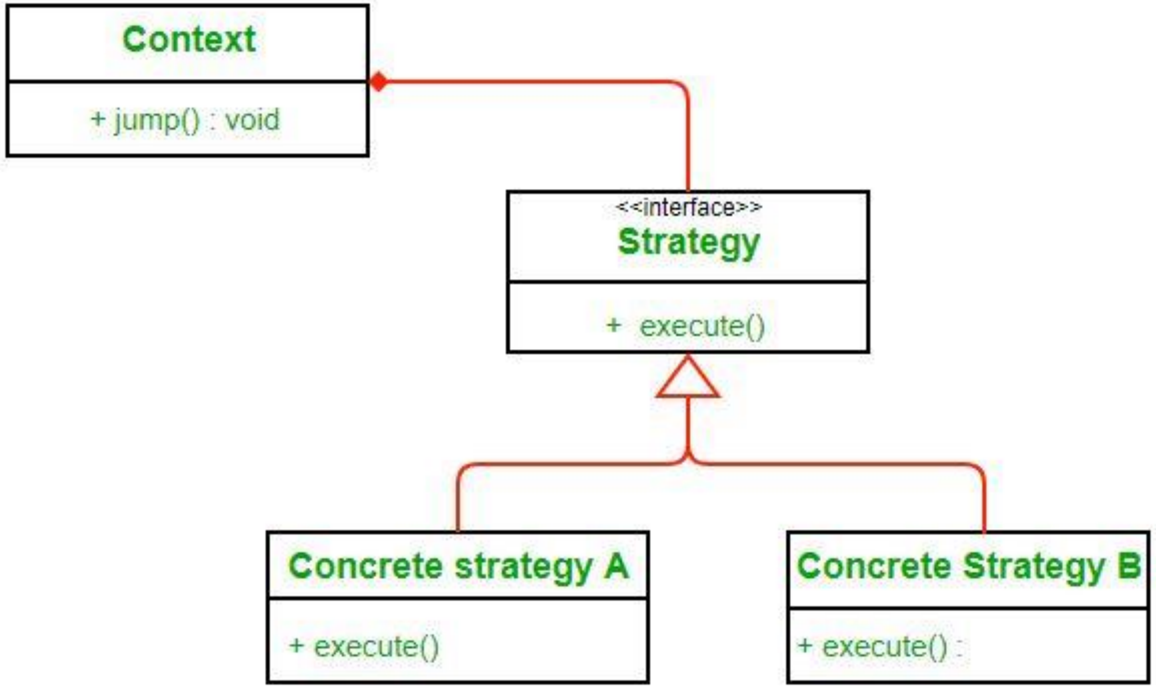
- Software design pattern that enables algorithm’s behavior to be selected at runtime
- The strategy pattern
  - Defines a family of algorithms
  - Encapsulates each algorithm
  - Makes algorithms interchangeable within that family

Strategy pattern



- Context of our problem is we need a way to have an optional jump
- Instead of implementing the behavior within the jump context, delegate that task to a Strategy

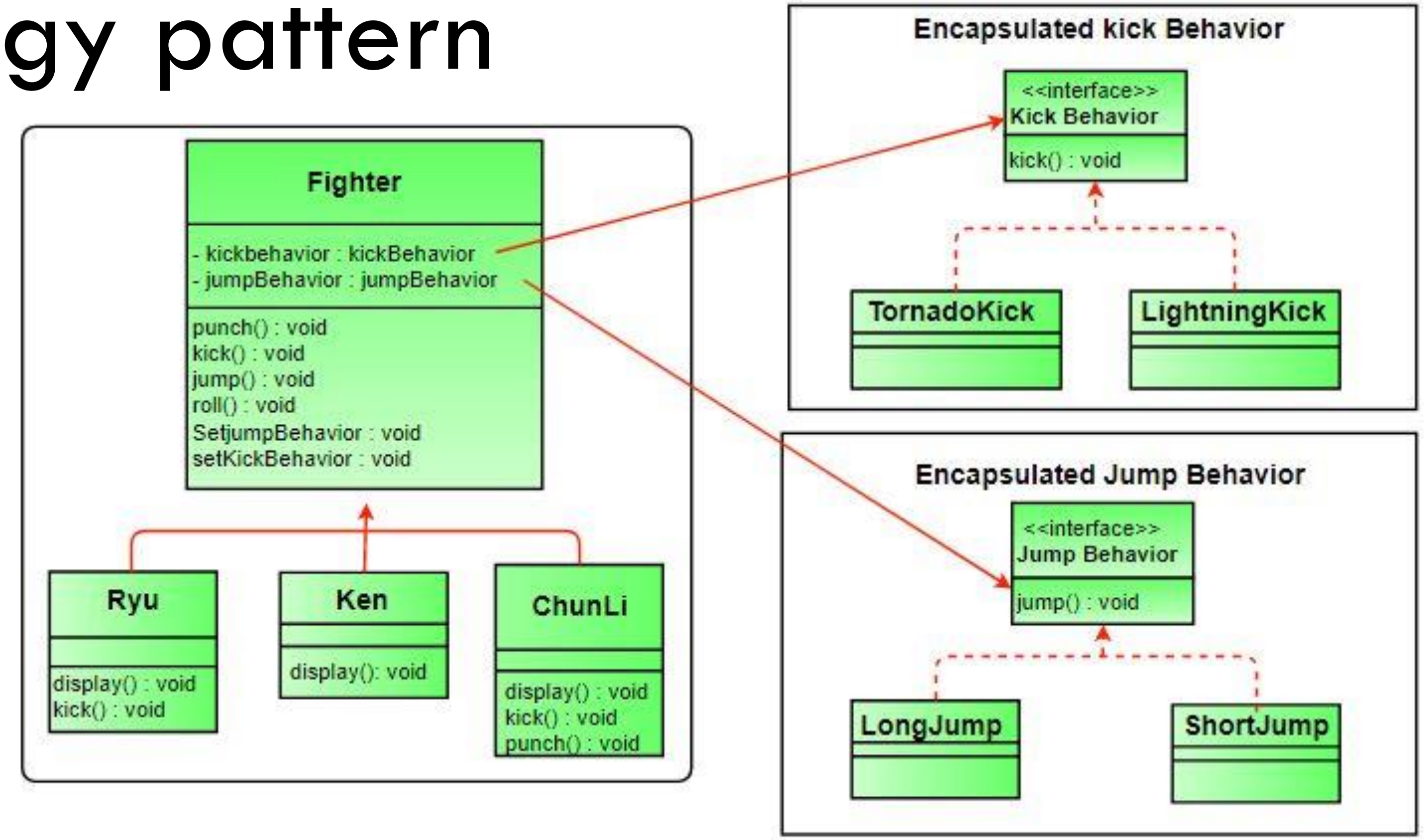
Strategy pattern



- Strategy is an interface that contains a function to execute the desired behavior
- The actual implementation of the behavior is done in the concrete classes

https://media.geeksforgeeks.org/wp-content/uploads/boxi3.jpg

Strategy pattern



- Pull out optional behaviors from Fighter
- Behaviors have their functionality defined in separate interfaces and subclasses
- Kick and Jump functions call kick and jump behaviors (interfaces)

Strategy pattern

https://www.geeksforgeeks.org/strategy-pattern-set-2/

*Java code

```java
// let us make some behaviors first
JumpBehavior shortJump = new ShortJump();
JumpBehavior LongJump = new LongJump();
KickBehavior tornadoKick = new TornadoKick();

// Make a fighter with desired behaviors
Fighter ken = new Ken(tornadoKick,shortJump);
ken.display();

// Test behaviors
ken.punch();
ken.kick();
ken.jump();

// Change behavior dynamically (algorithms are
// interchangeable)
ken.setJumpBehavior(LongJump);
ken.jump();
```

- In main, construct behaviors first
- Add optional behaviors to fighter
- Fighter can perform default and optional moves
- Fighter can even swap out and execute behavior at runtime

strategy.cpp

lvalue AND rvalue

C++ expressions

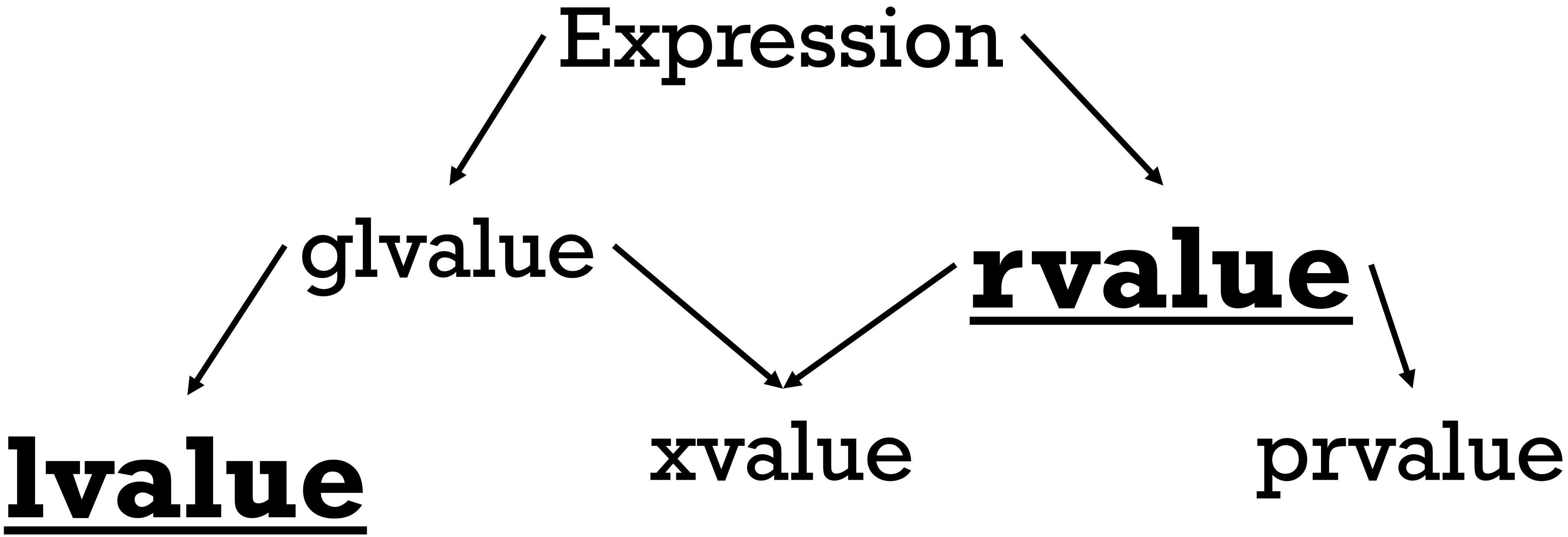
- Expression - sequence of operators and operands that specifies a computation
- Examples of expressions include:
    - An operator with its operands (x + y)
    - A literal (int, char, floating point, string, boolean, etc.) (5, ‘c’, 2.0, “hello”, true, etc)
    - A variable name or identifier, etc.
- **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

```
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

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`

ACTIVITY

1. Examine the following code and discuss with your neighbours:
    1. lvaluervalue.cpp
    2. lvaluervalue2.cpp
    3. lvaluervalue3.cpp
    4. lvaluervalue4.cpp
    5. lvaluervalue5.cpp

MOVE: CONSTRUCTOR AND OPERATOR

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

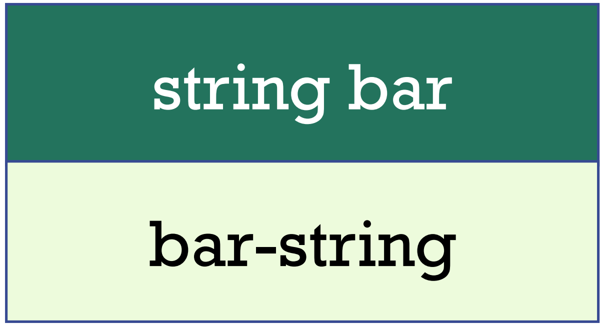
What happens to the “original”

- 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**

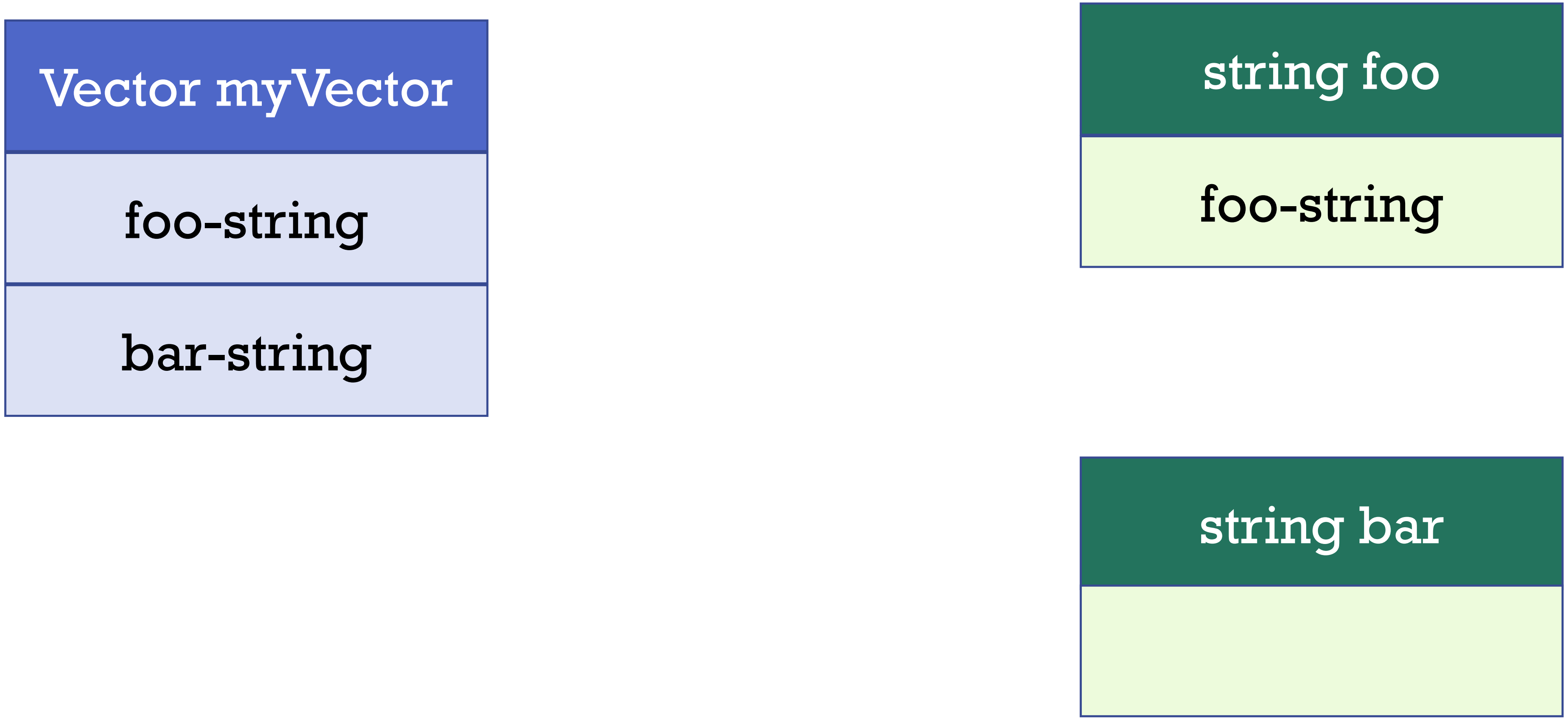
- Check out **move.cpp**

myvector.push_back (foo);

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



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



“bar-string” moved from bar and pushed into myVector

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**

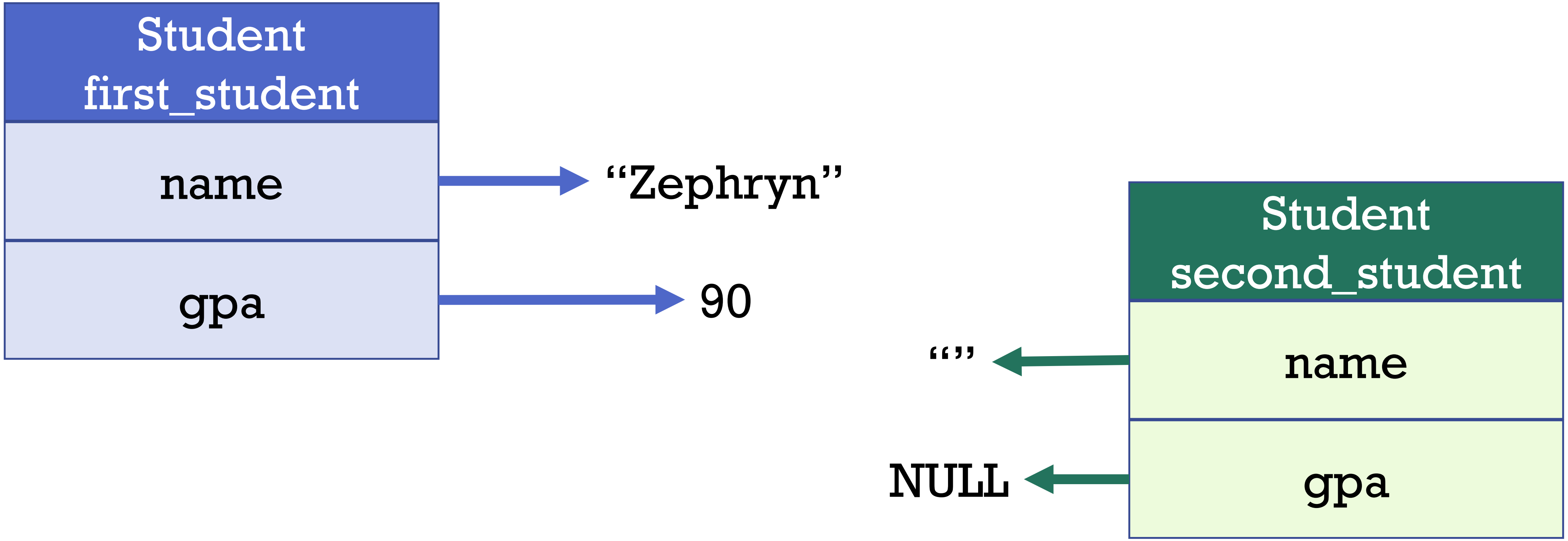
Move constructor

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

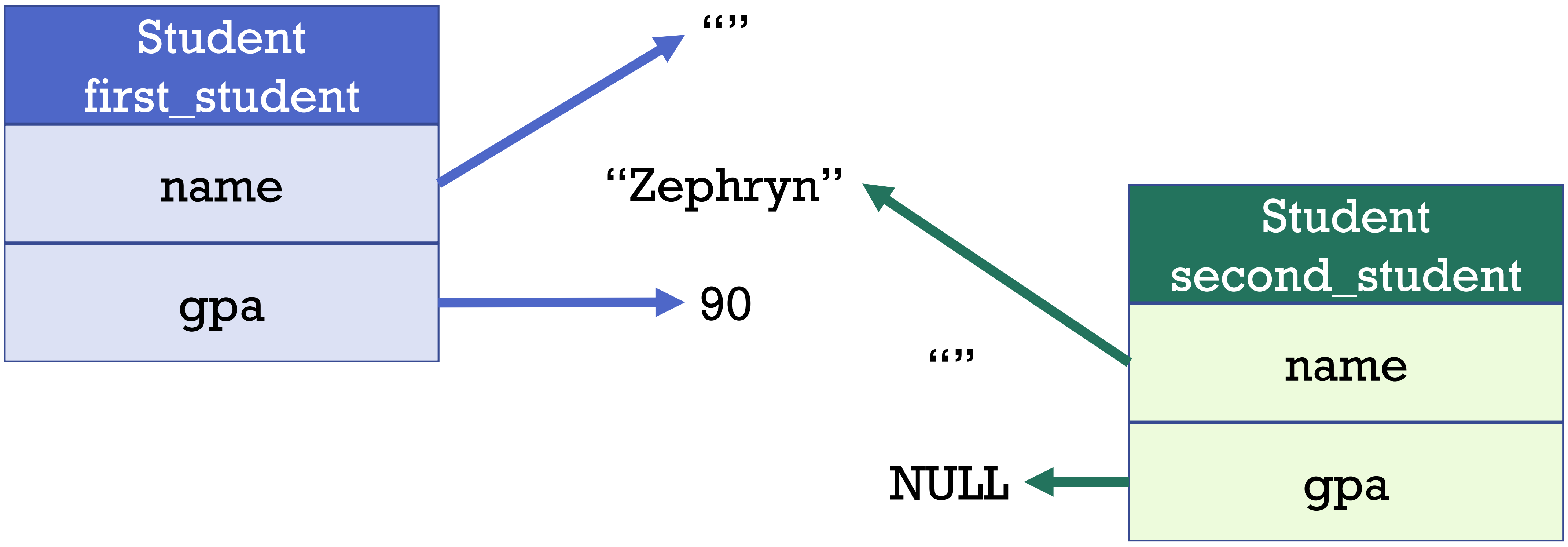
- Takes ownership of member variables from another object
- Faster, avoids memory allocation (unlike copy constructor)
- Kind of "shallow copy"
- Check out **move2.cpp** and **move3.cpp**

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

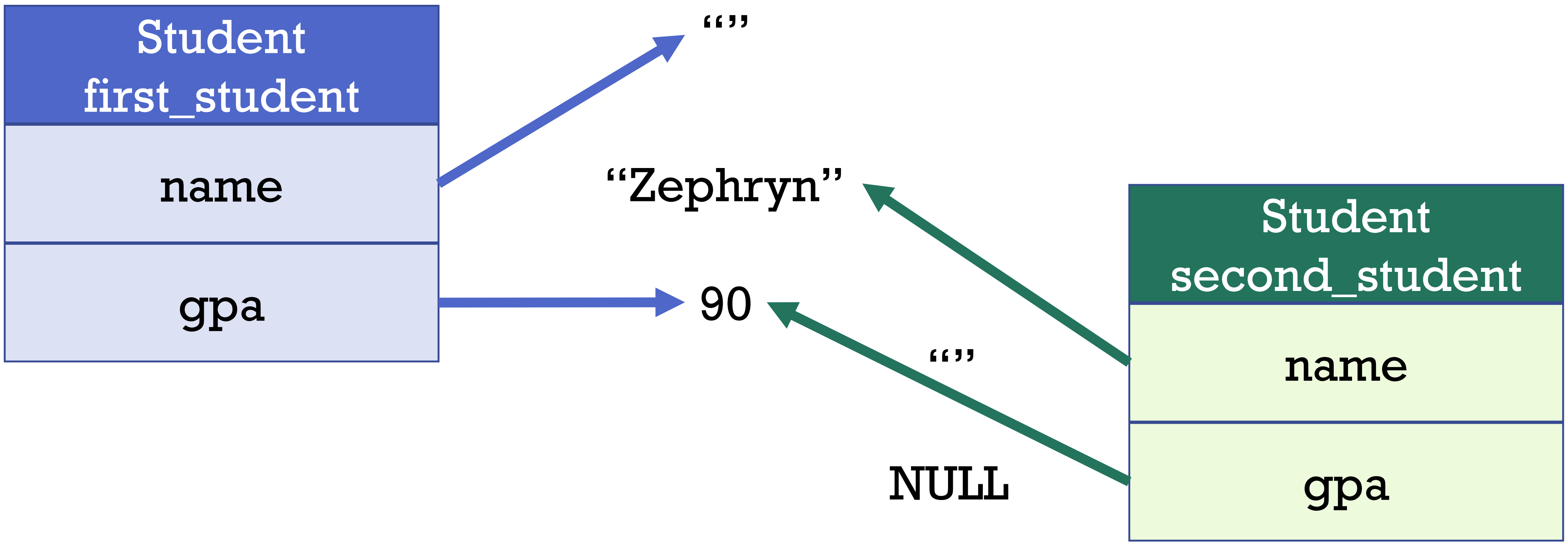
```
student second_student(std::move(first_student));
```



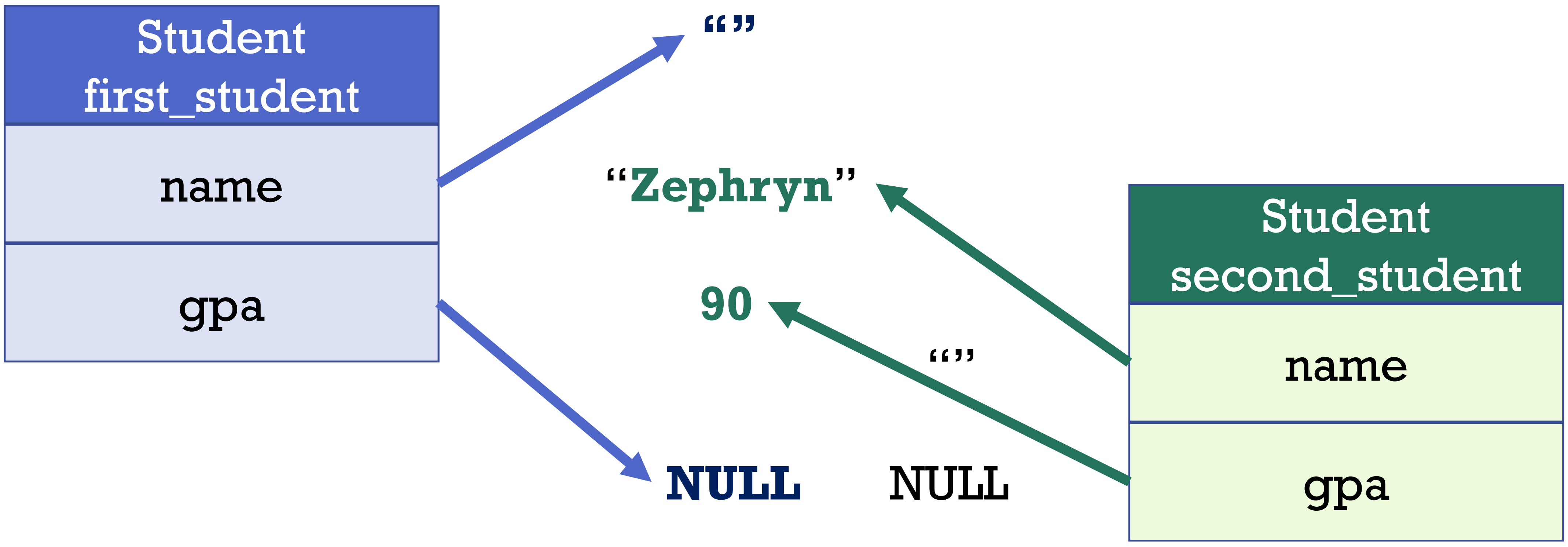
```
student(student&& other) : name{move(other.name)}
{
        gpa = move(other.gpa);
        other.gpa = nullptr;
}
student second_student(std::move(first_student));
```



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



```
student(student&& other) : name{move(other.name)}
{
        gpa = move(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 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.

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!

- But now we can’t really respond to dynamic user input, or do other fun, ‘dynamic’ things

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
- Primarily used in rare cases to break circular references, i.e., in doubly linked lists
- Check out **weak_ptr_1.cpp**, **weak_ptr_2.cpp**, **weak_ptr_3.cpp**, **weak_ptr_4.cpp**, **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