移动语义基础

Move Semantics Basics

现代C++基础

Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

- Part 1
- Introduction

- Move Ctor & Assignment
  - Discussions on `noexcept` and inheritance.

- Rule of Zero/Five

- Moved-from States

- Algorithms and Iterators with Move Semantics

Value Category and Move Semantics

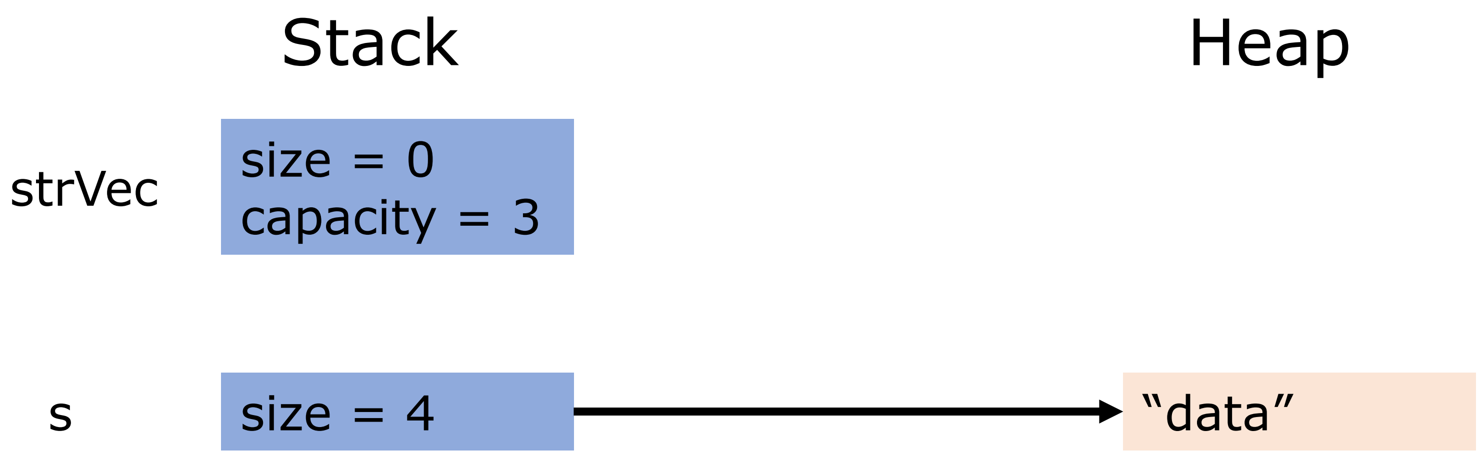
Introduction

Why move semantics?

- Let’s start demystifying move semantics by an example!
  - We omit SSO for `std::string`, **so characters are all allocated on heap.**
- During the slides, you can observe whether some operations are in fact unnecessary.

Why move semantics?

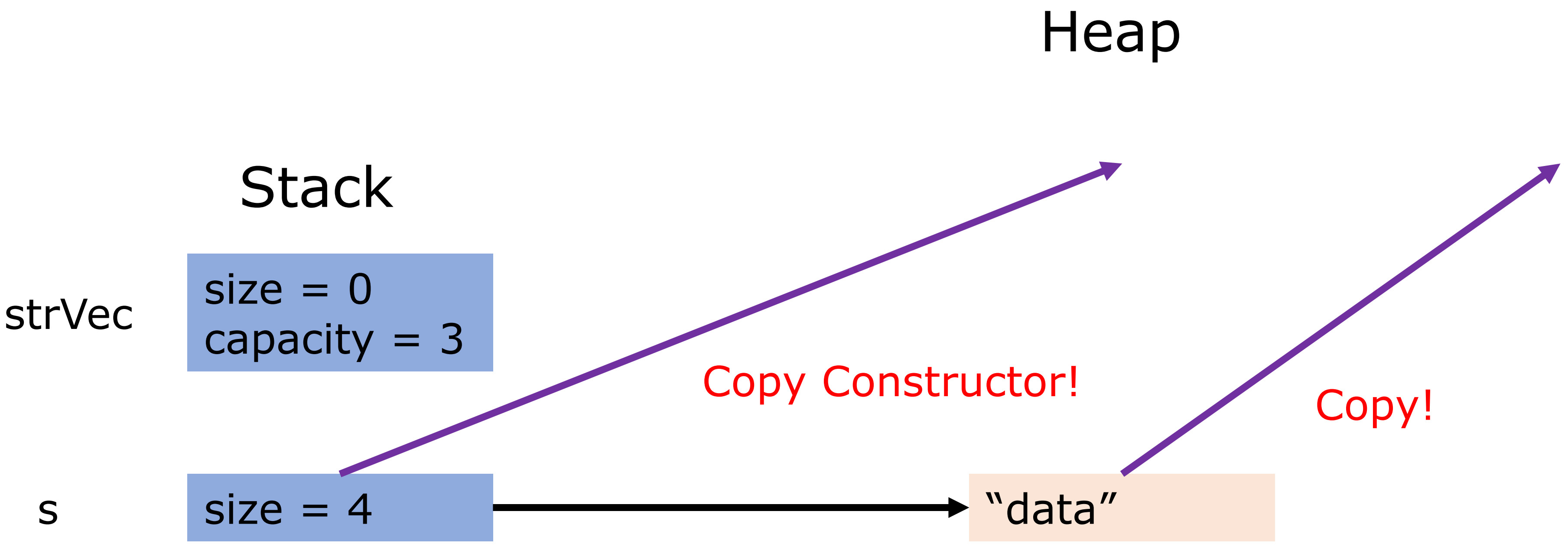
- In function `CreateAndInsert()`, we know the memory layout before `push_back` should be like:



- First observe how C++98 (no move semantics) works.

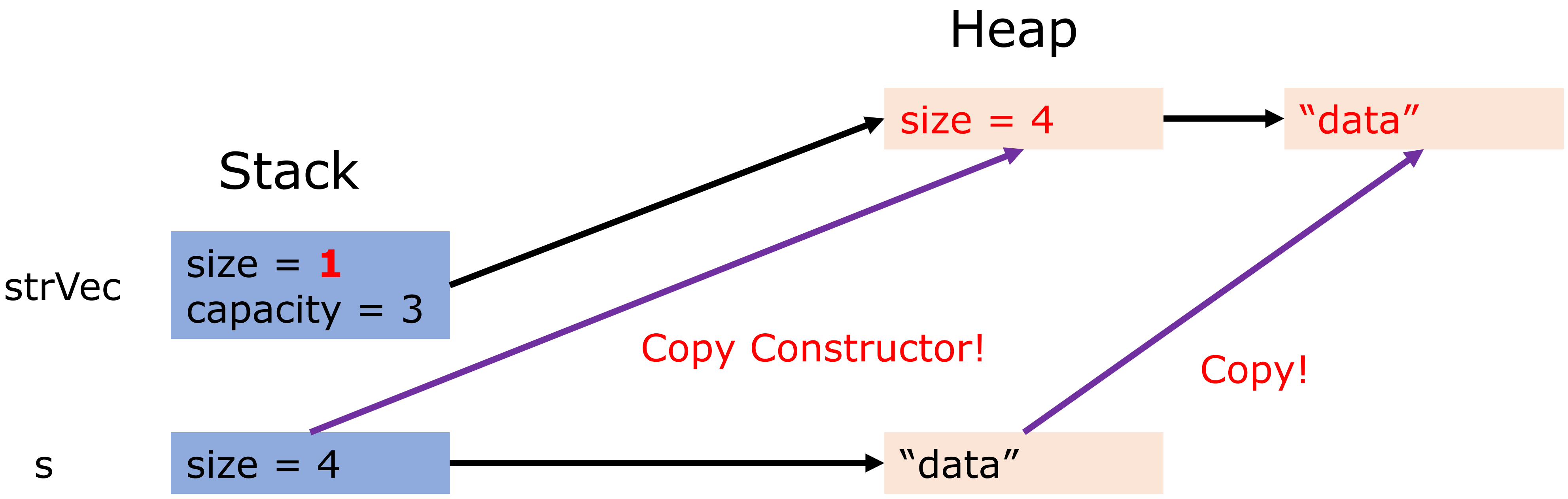
Why move semantics?

- `strVec.push_back(s);`



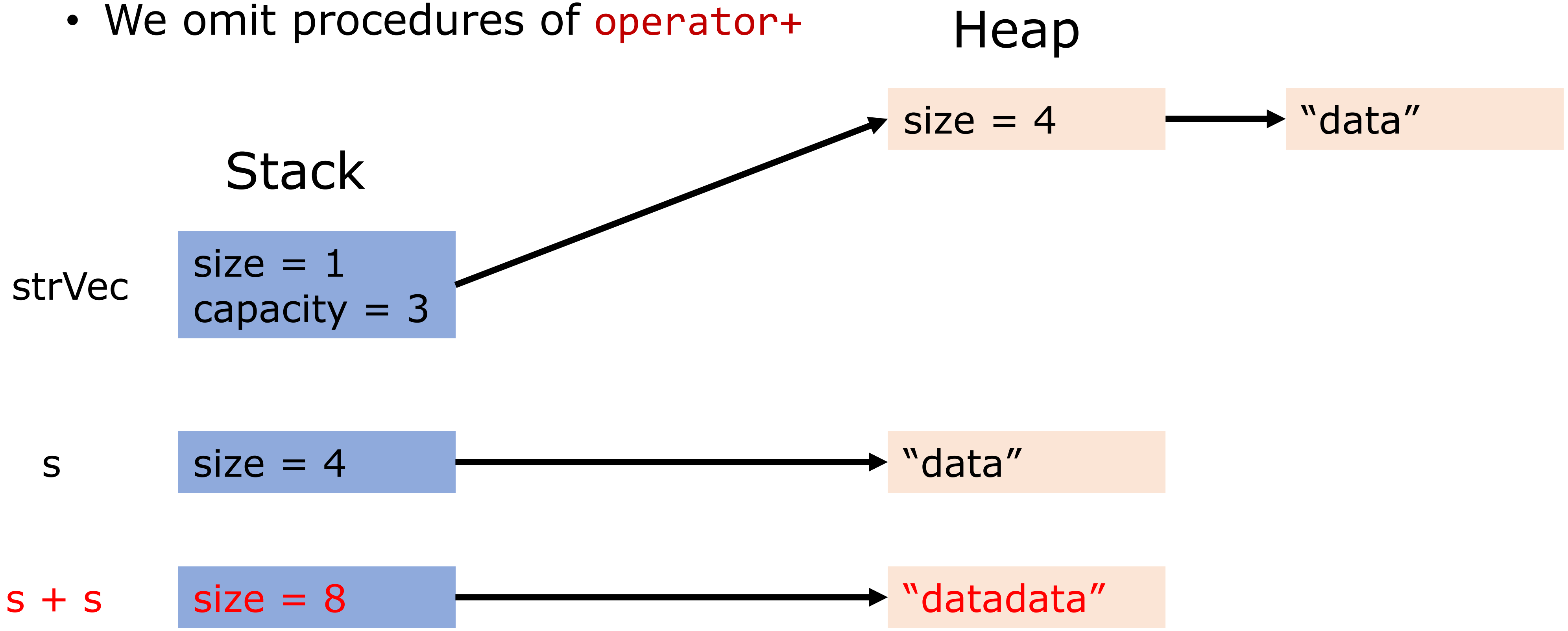
Why move semantics?

- `strVec.push_back(s);`



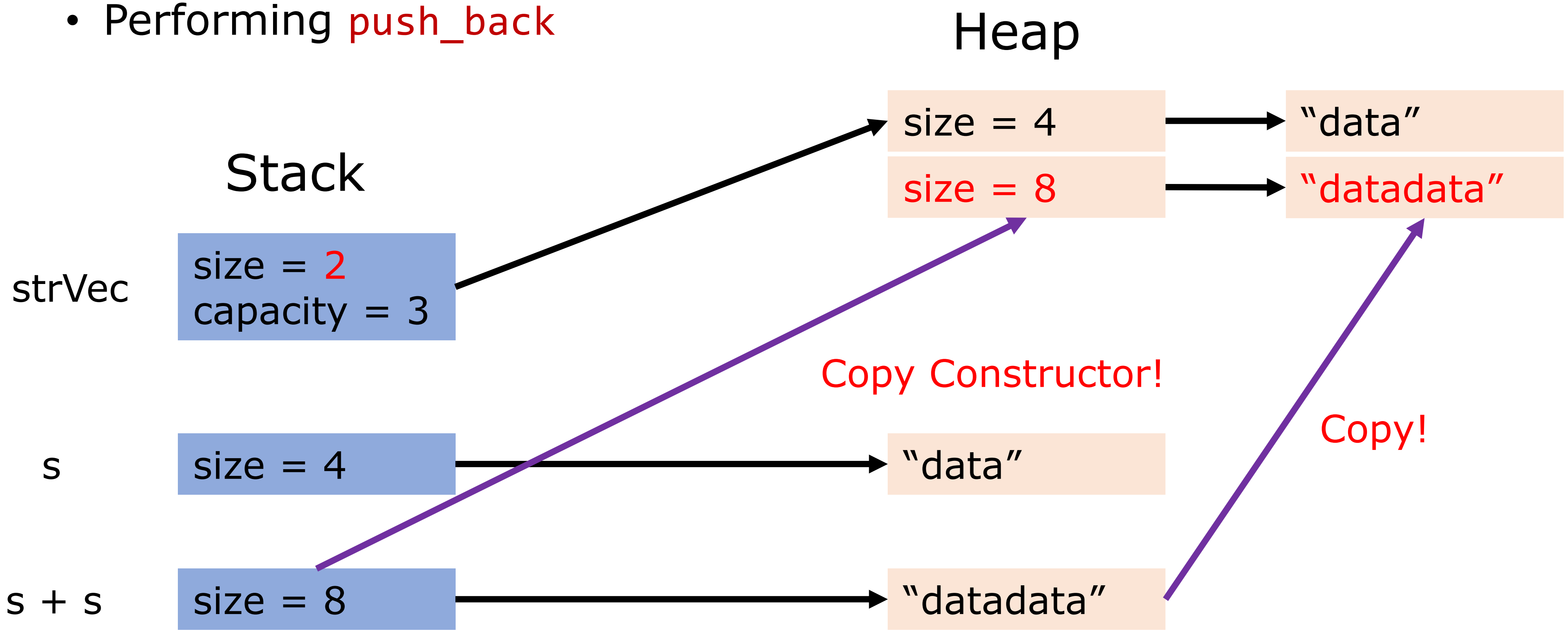
Why move semantics?

- `strVec.push_back(s+s);`
  - We omit procedures of `operator+`



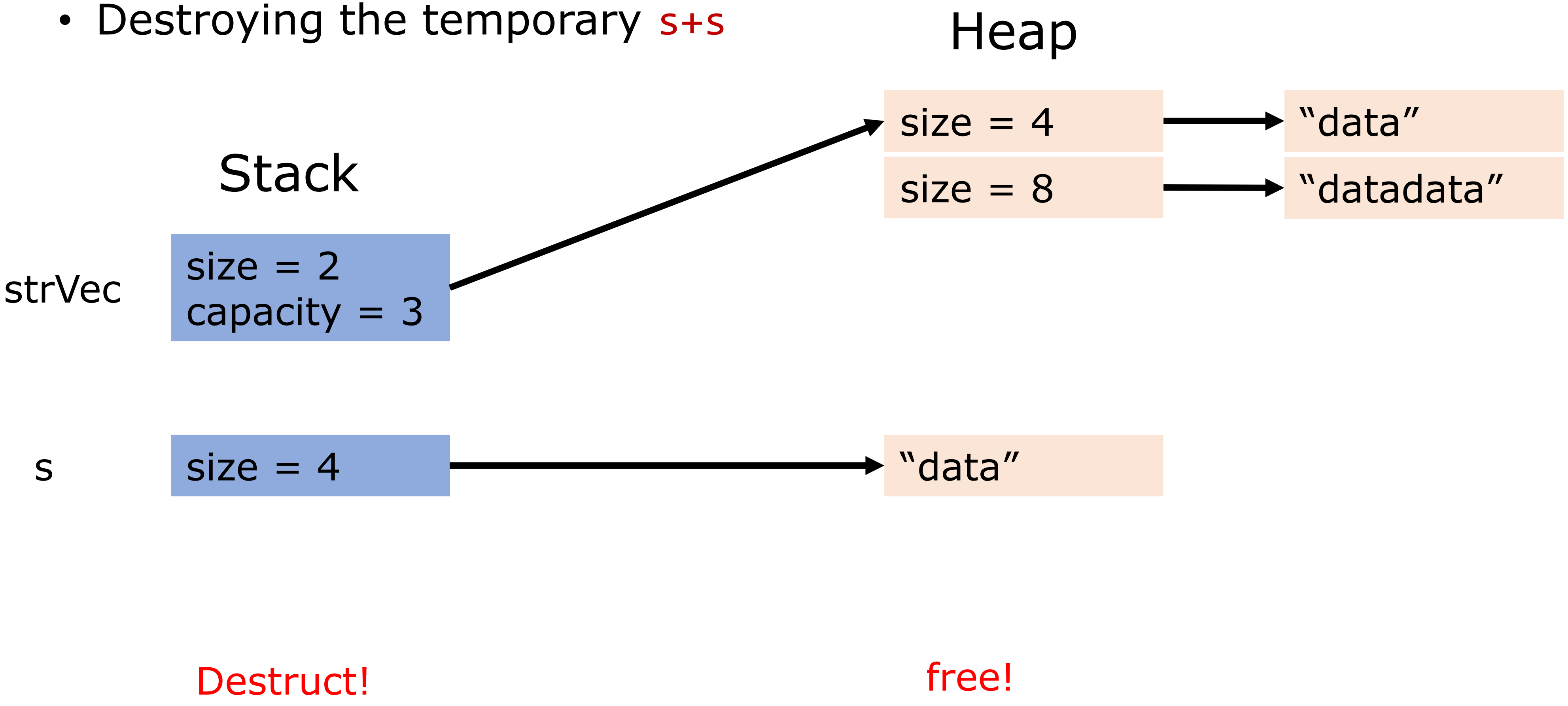
Why move semantics?

- `strVec.push_back(s+s);`
  - Performing `push_back`



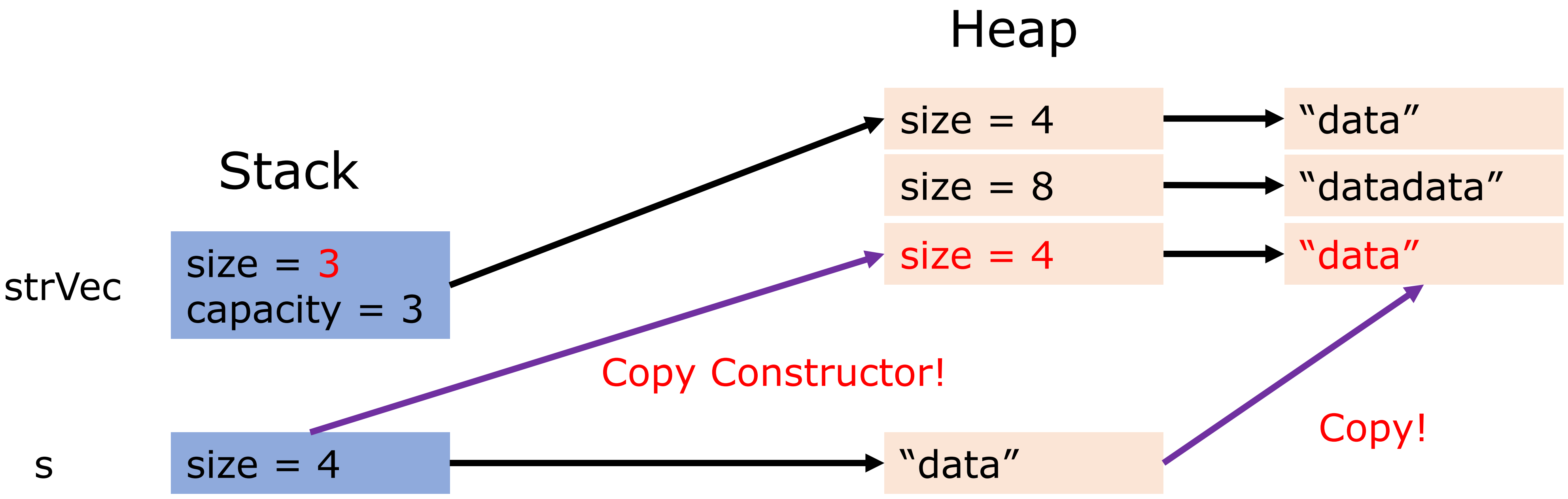
Why move semantics?

- `strVec.push_back(s+s);`
  - Destroying the temporary `s+s`



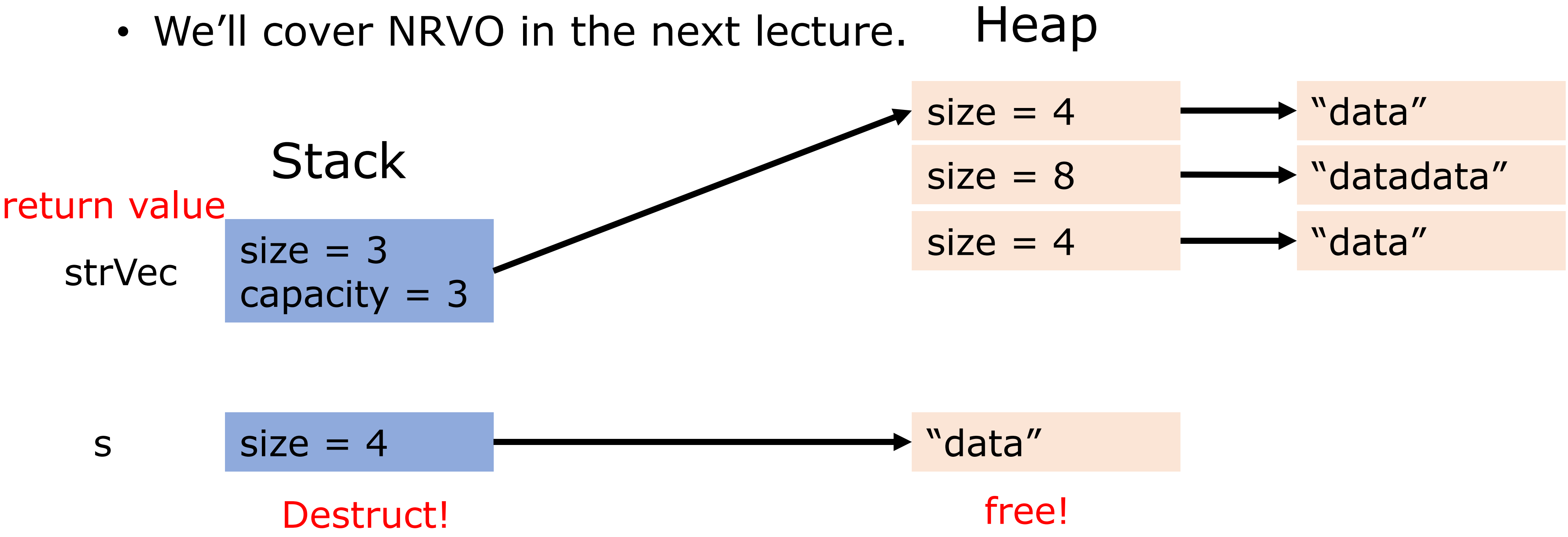
Why move semantics?

- `strVec.push_back(s);`



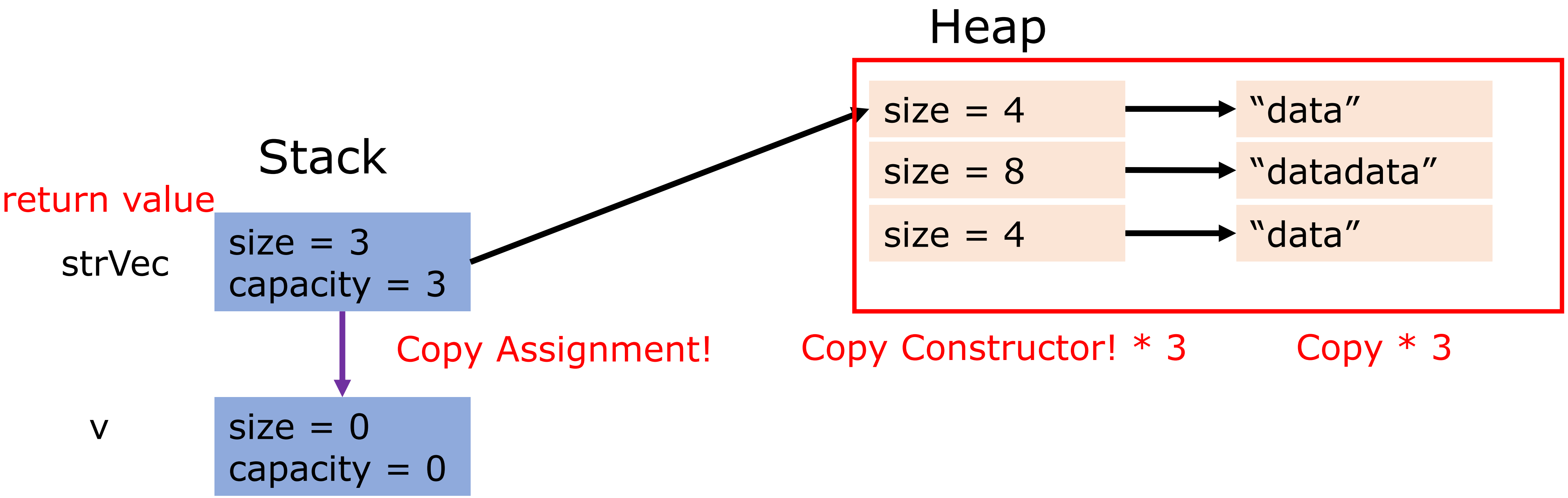
Why move semantics?

- `return strVec;`
  - NRVO avoids creating a new temporary to return.
  - We'll cover NRVO in the next lecture.



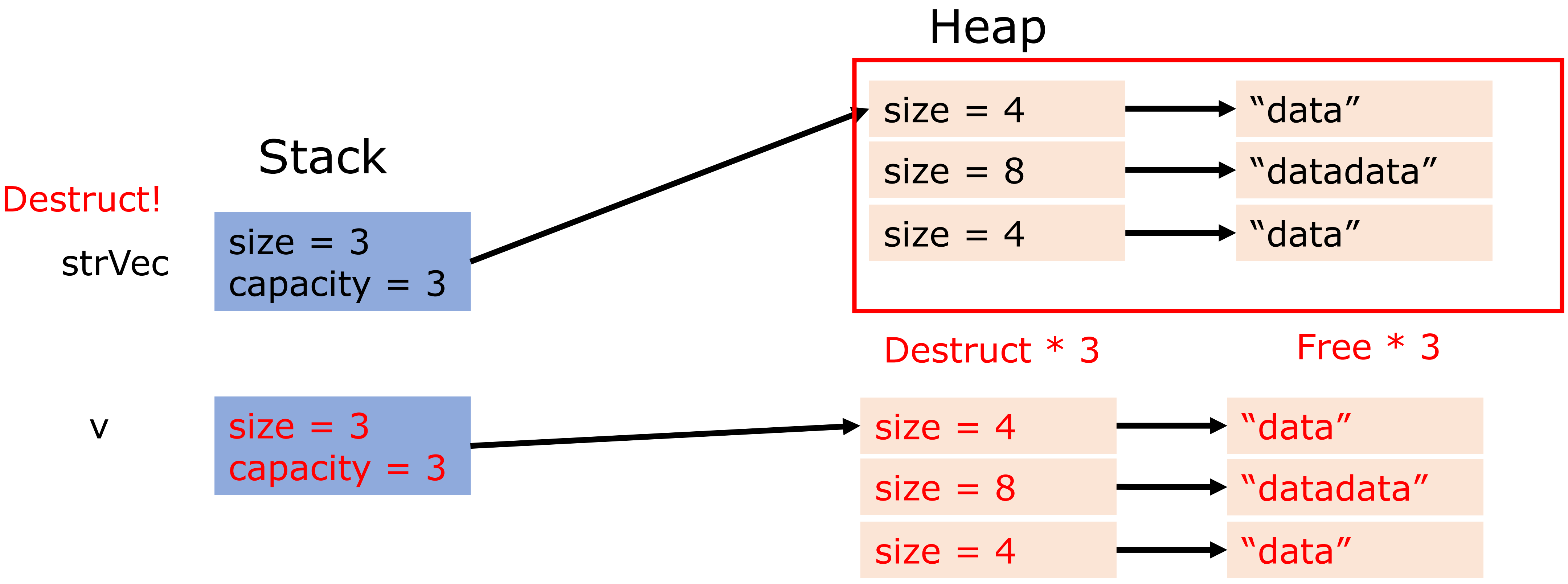
Why move semantics?

- `v = CreateAndInsert();`



Why move semantics?

- `v = CreateAndInsert();`



Why move semantics?

- Totally, we call copy ctor for 6 times & dtor for 5 times, leading to copy × 6 and free × 5.
- Key observation
  - Some resources are copied and then released!
  - e.g. `strVec.push_back(s+s);`
  - We can “move” them directly to avoid both copying and releasing!

Why move semantics?

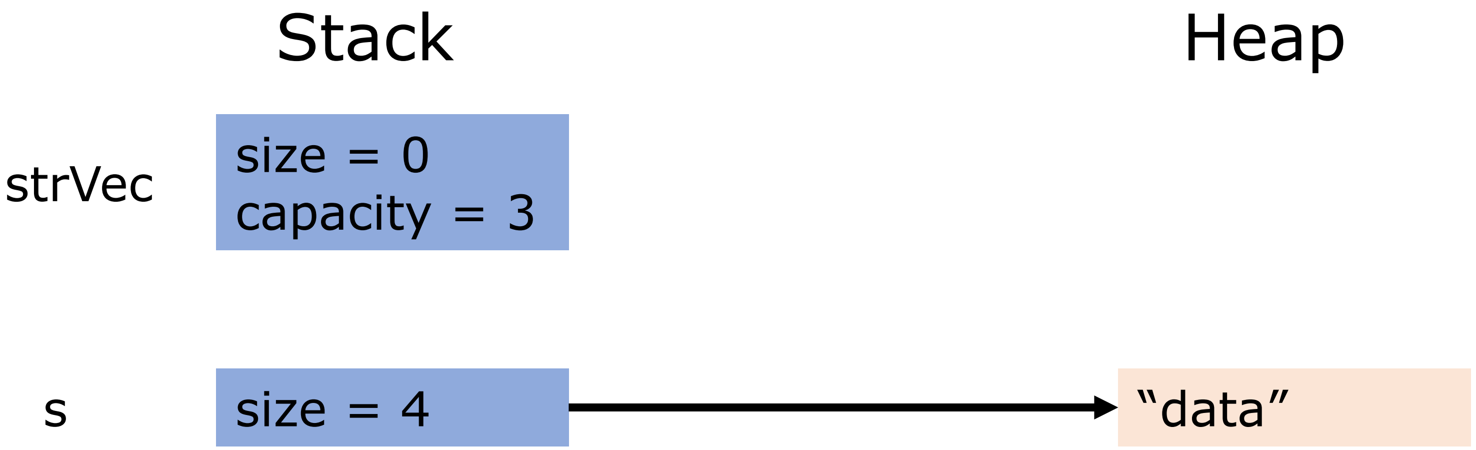
- So, let's try C++11 with move semantics.
- We first make code utilize move semantics more:

```cpp
#include <vector>
#include <string>

std::vector<std::string> CreateAndInsert()
{
    std::vector<std::string> strVec;
    strVec.reserve(3); // prohibit reallocation to reduce complexity.
    std::string s = "data";

    strVec.push_back(s);
    strVec.push_back(s + s);
    strVec.push_back(std::move(s));

    return strVec;
}

int main()
{
    std::vector<std::string> v;
    v = CreateAndInsert();
    return 0;
}
```

The only change in code!

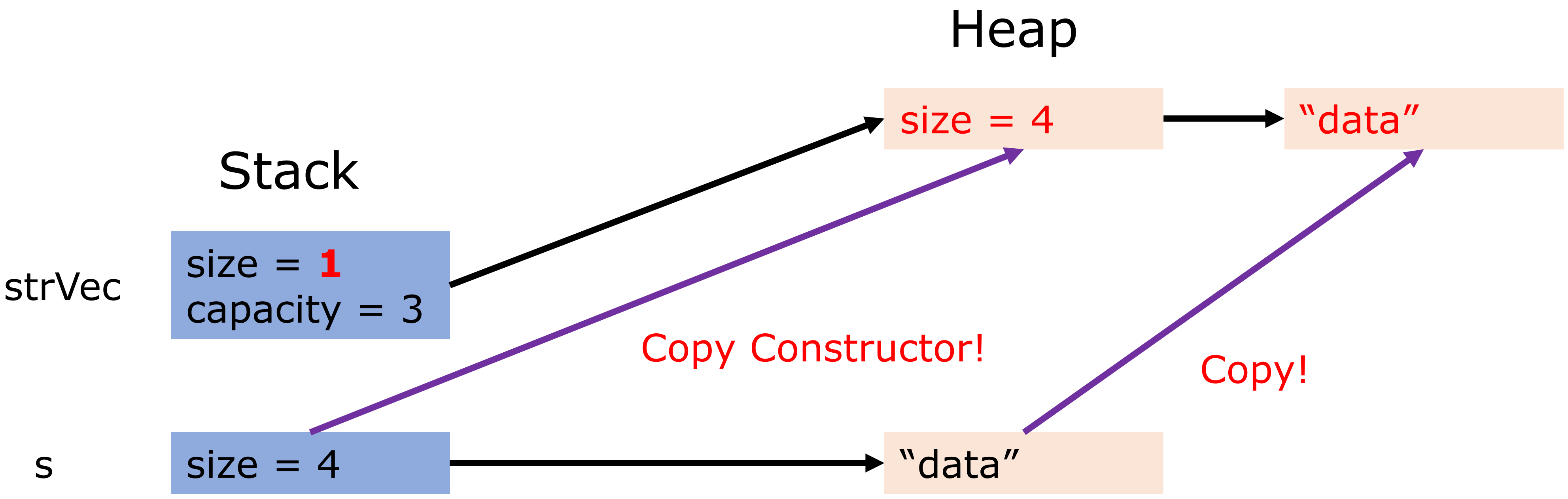
Why move semantics?

- Start from the scratch:



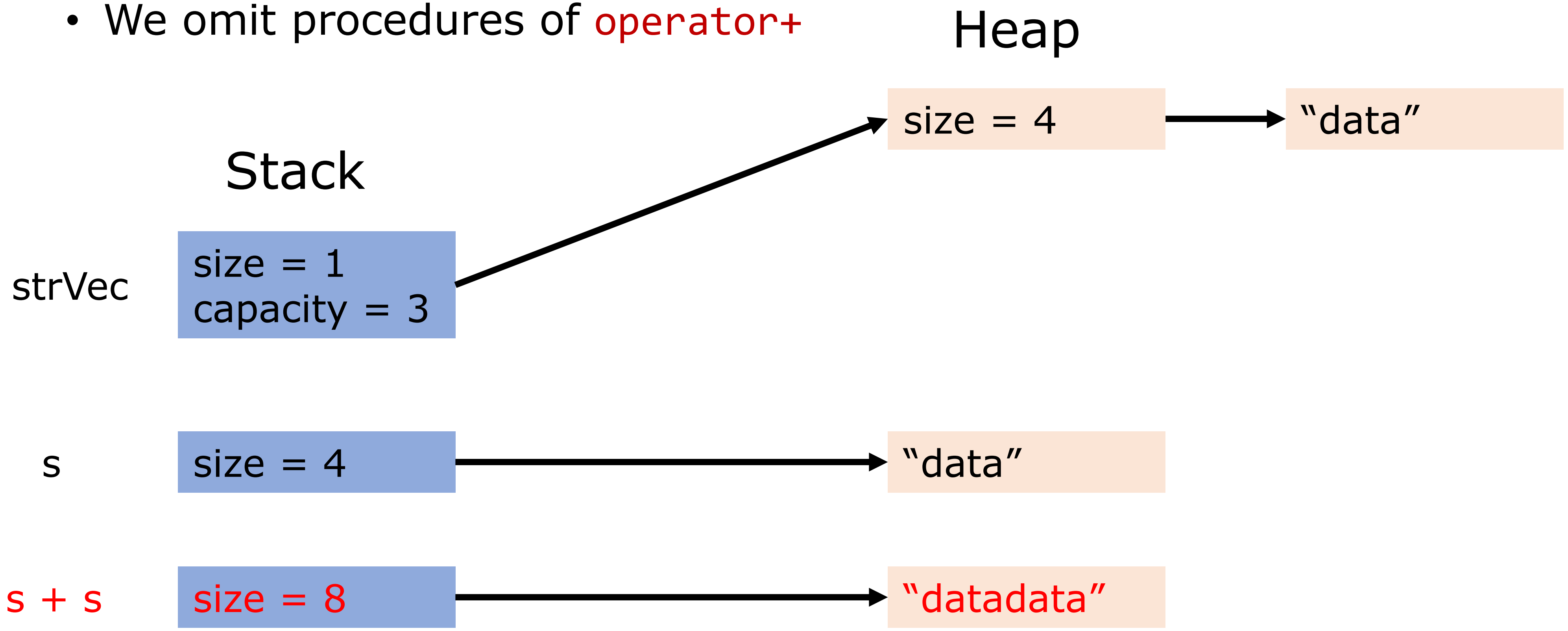
Why move semantics?

- `strVec.push_back(s);`



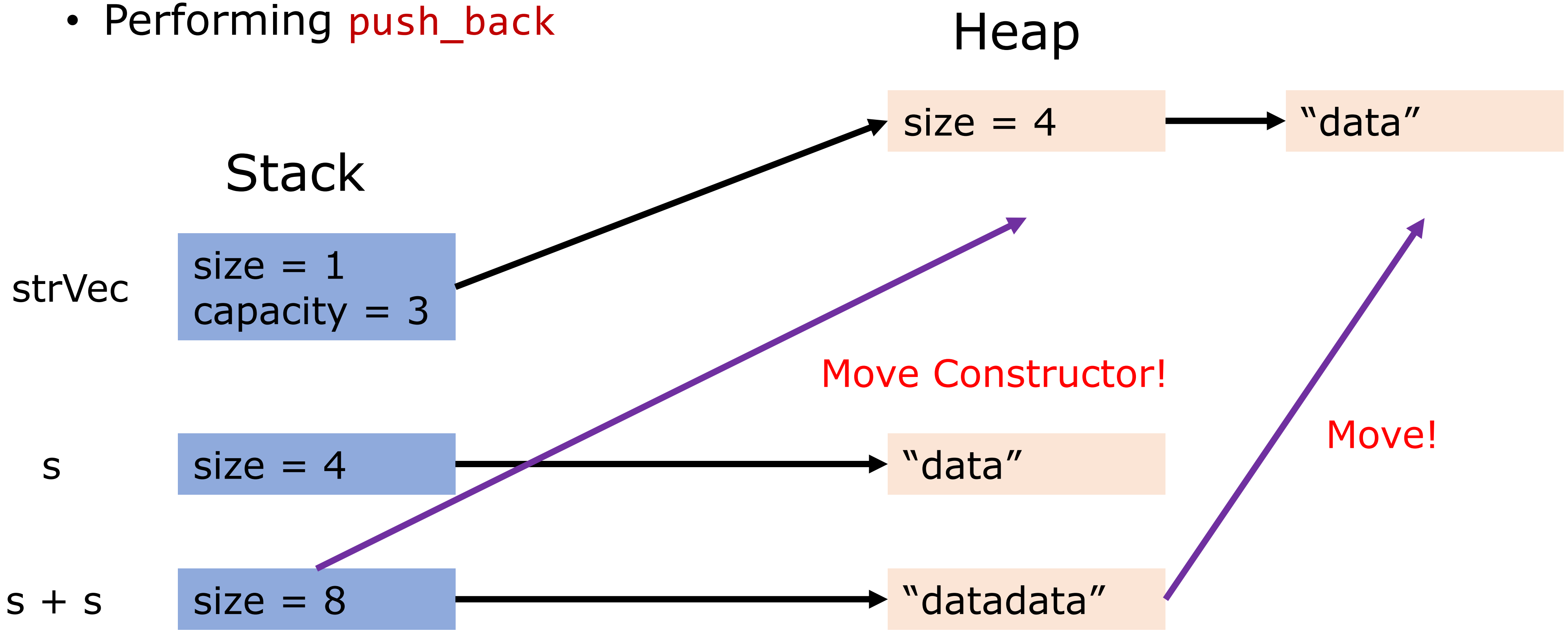
Why move semantics?

- `strVec.push_back(s+s);`
  - We omit procedures of `operator+`



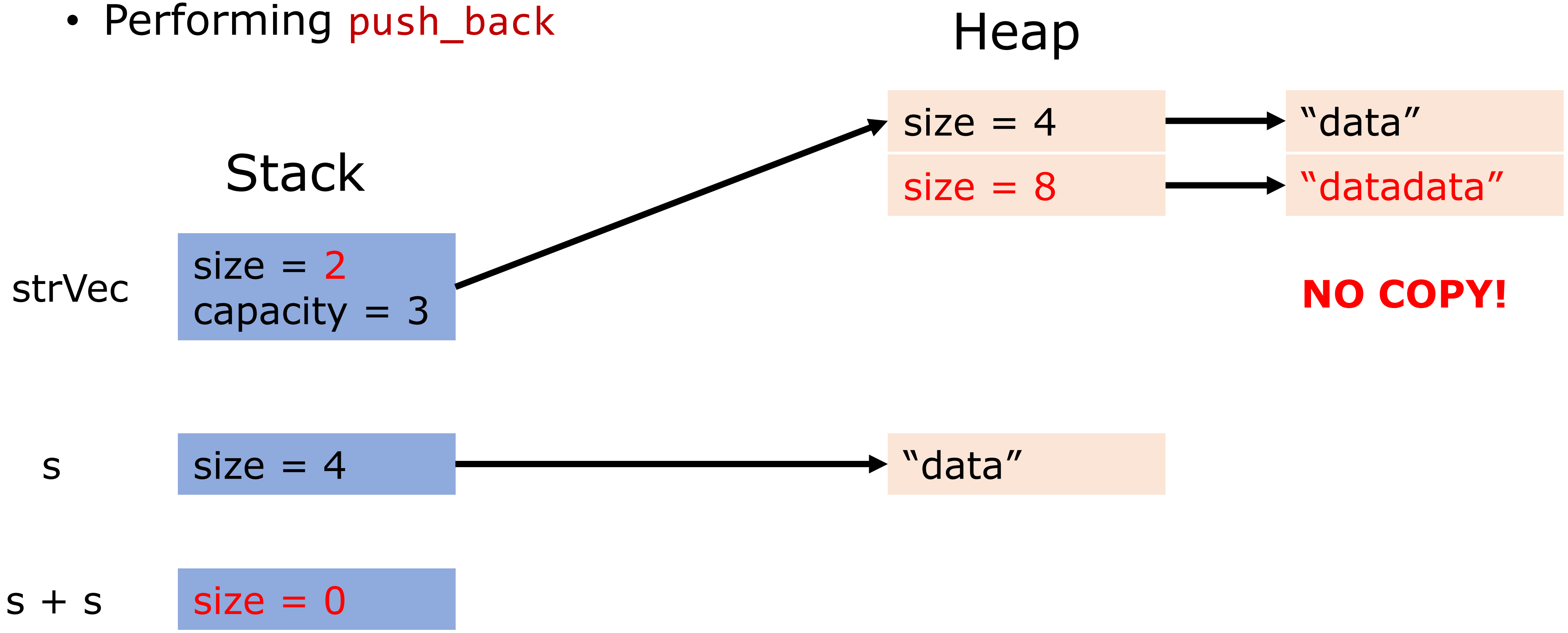
Why move semantics?

- `strVec.push_back(s+s);`
  - Performing `push_back`



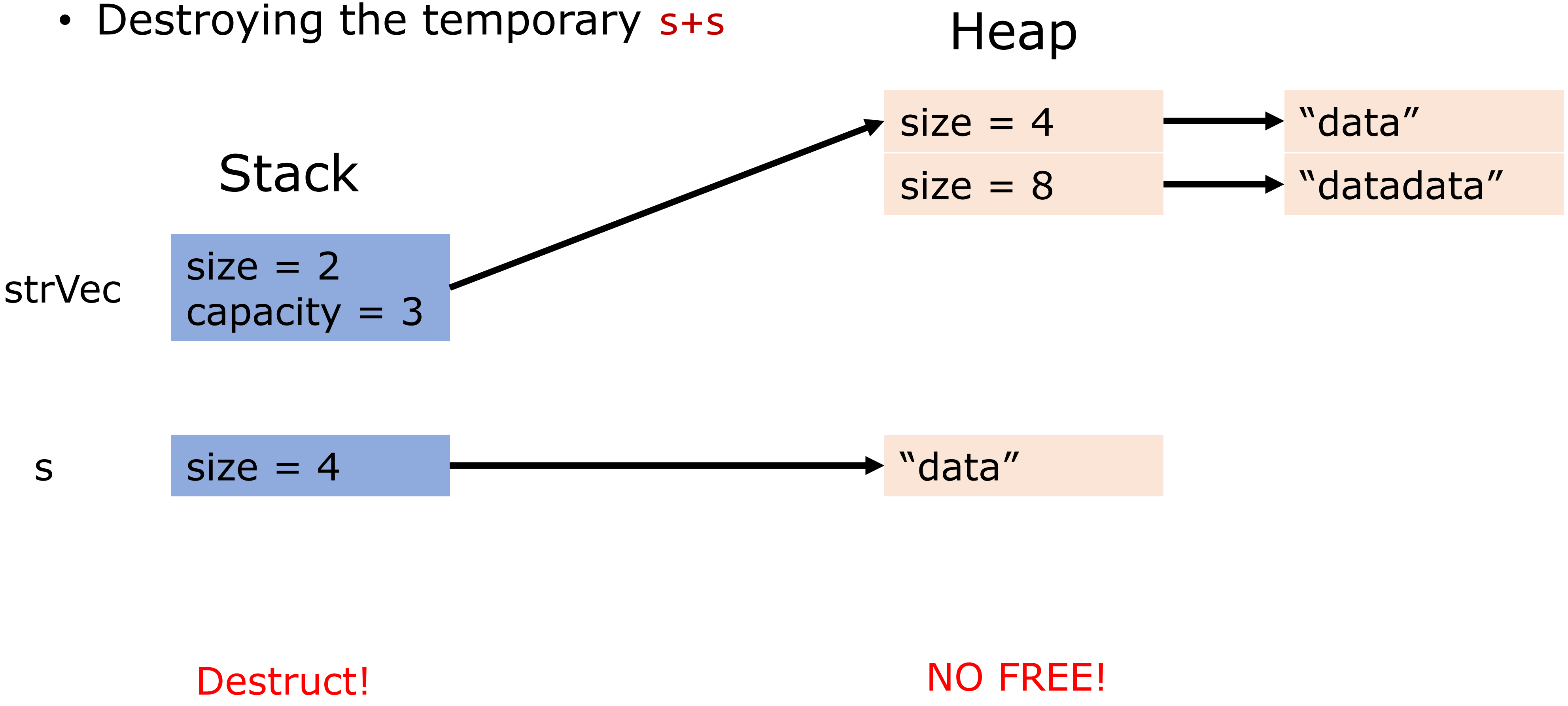
Why move semantics?

- `strVec.push_back(s+s);`
  - Performing `push_back`



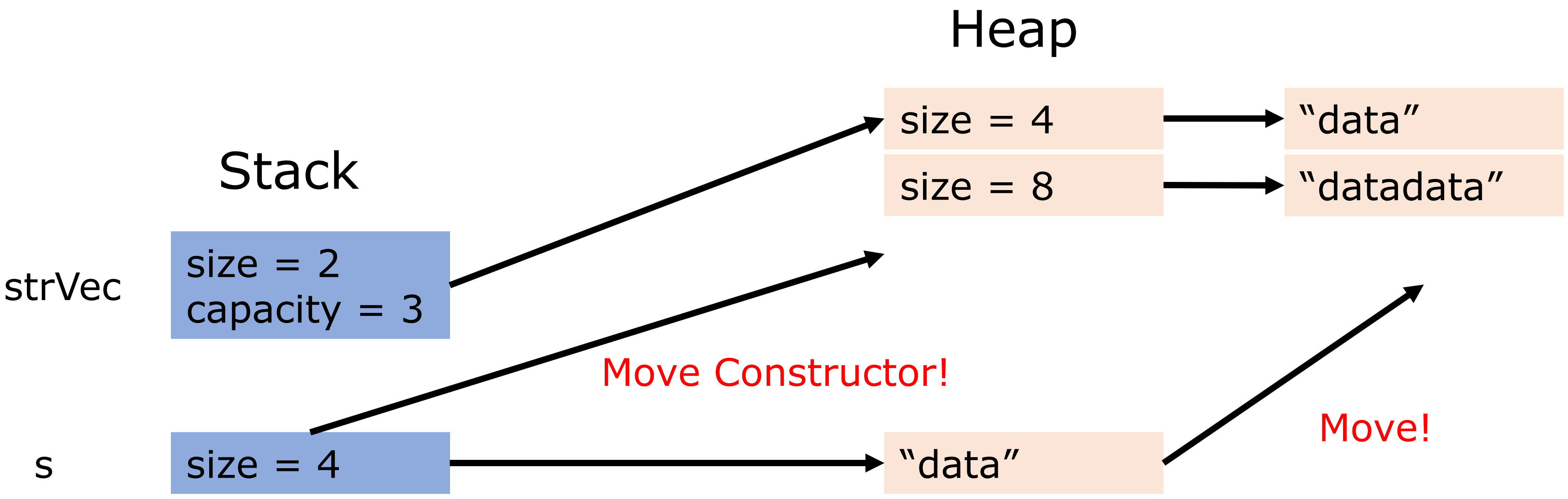
Why move semantics?

- `strVec.push_back(s+s);`
  - Destroying the temporary `s+s`



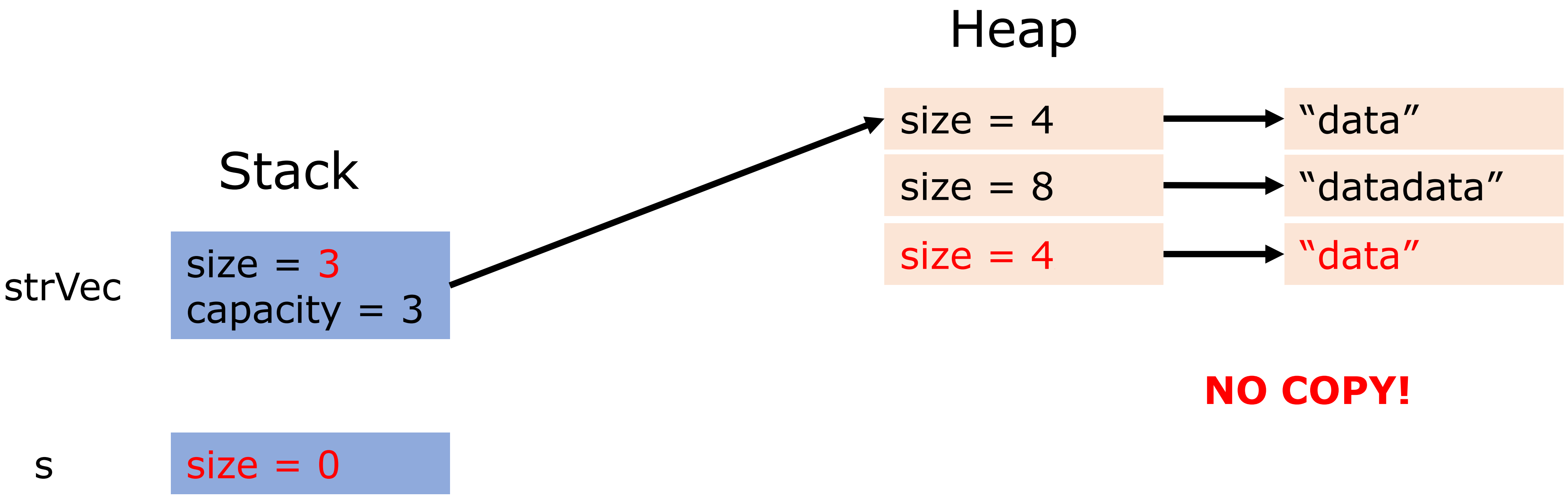
Why move semantics?

- `strVec.push_back(std::move(s));`



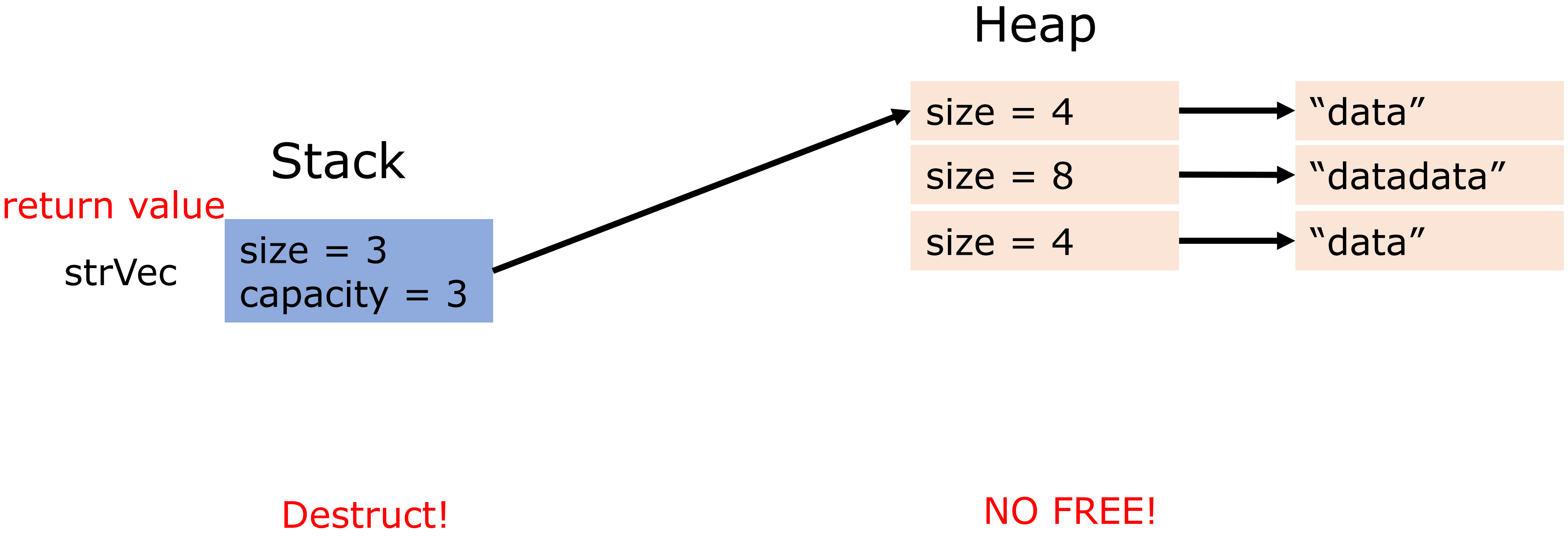
Why move semantics?

- `strVec.push_back(std::move(s));`



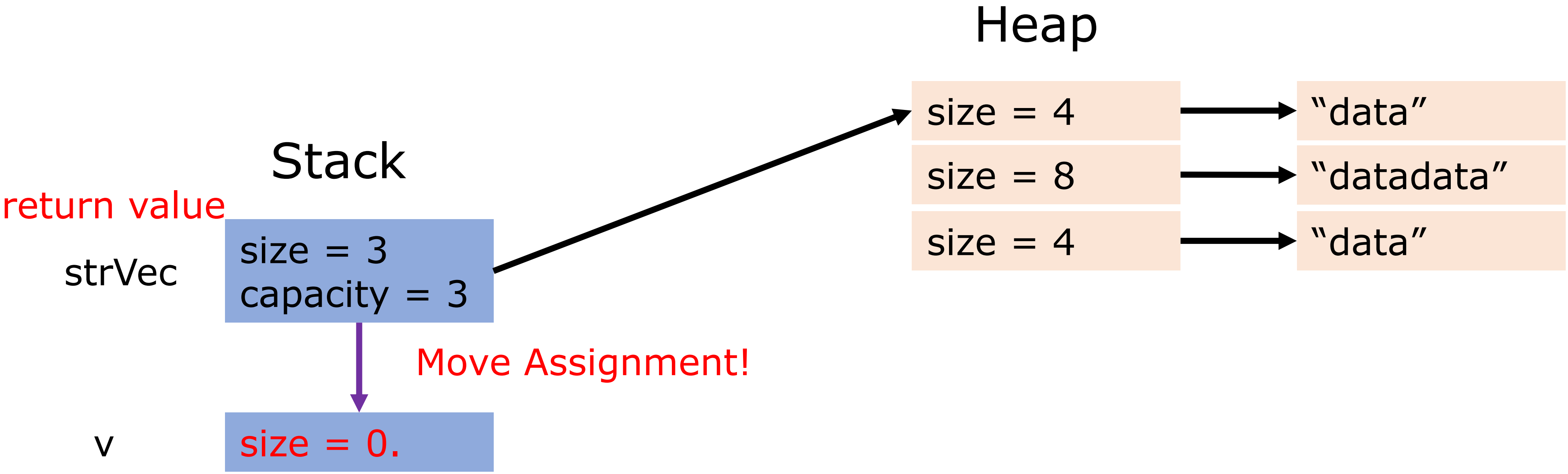
Why move semantics?

- `return strVec;`
  - NRVO avoids creating a new temporary to return.



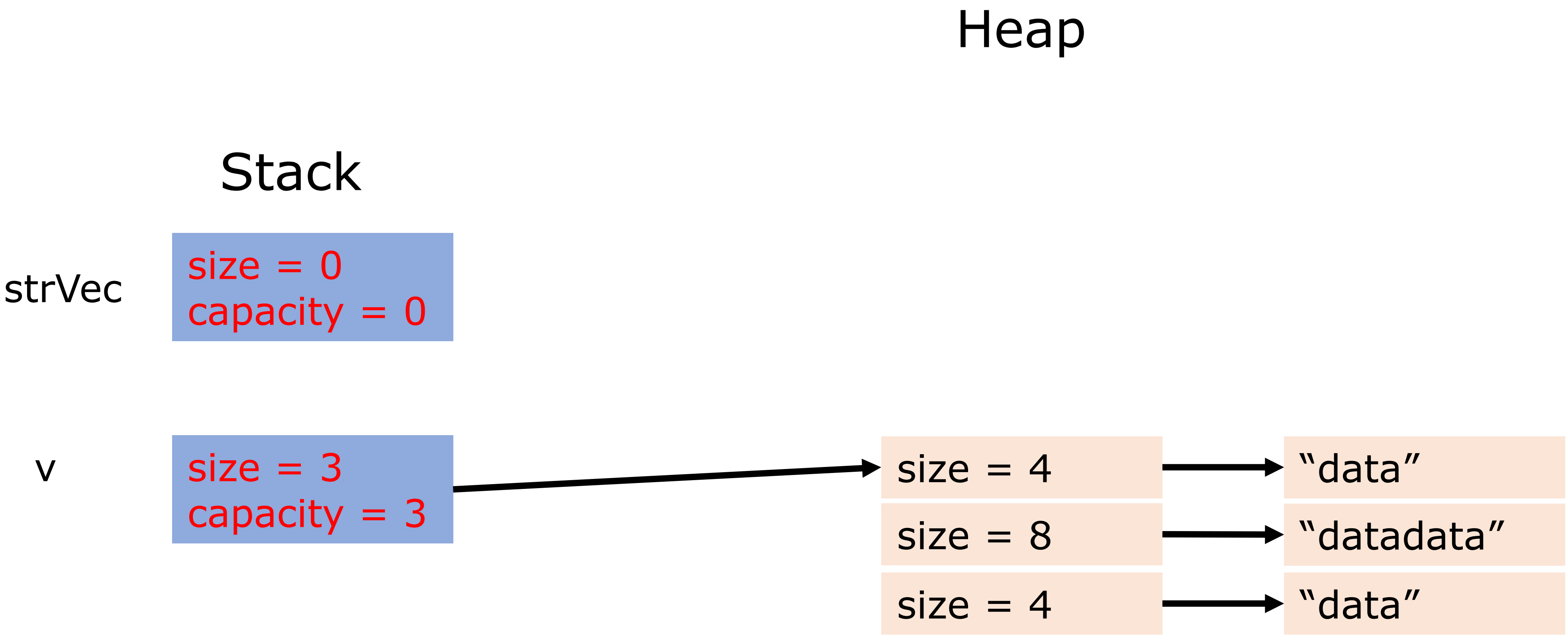
Why move semantics?

- `v = CreateAndInsert();`



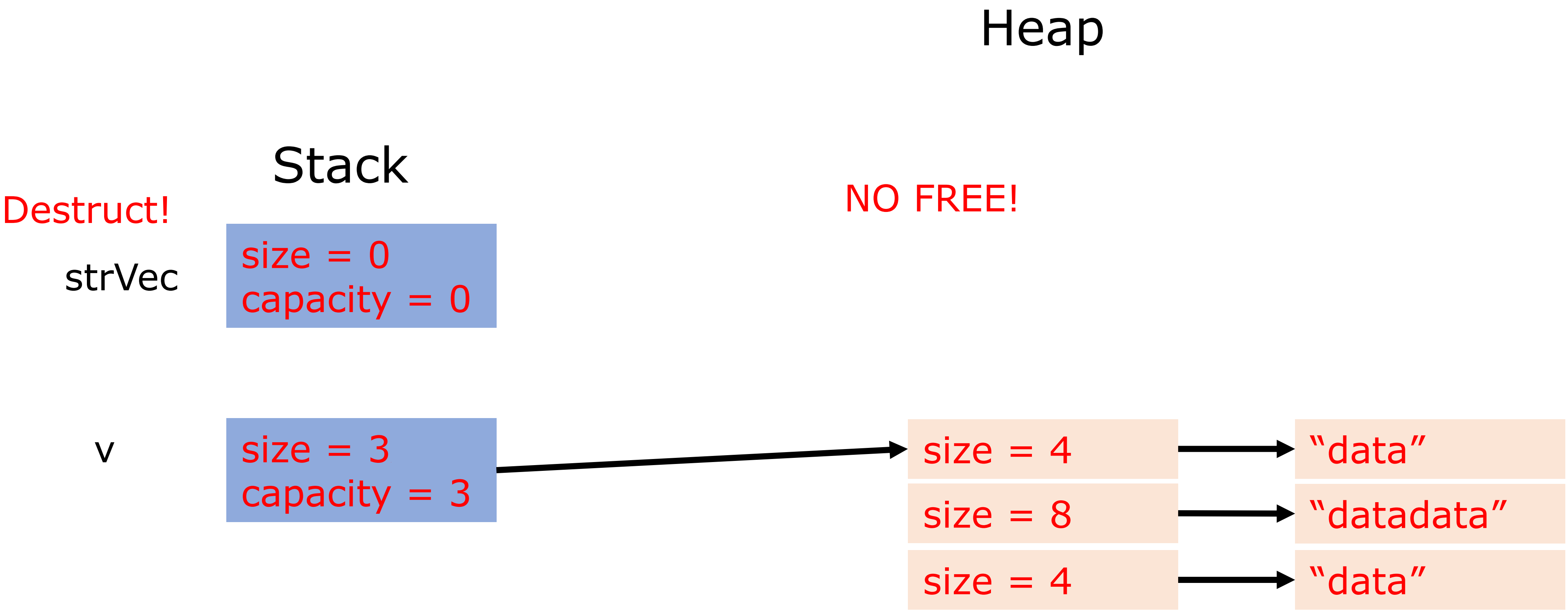
Why move semantics?

- `v = CreateAndInsert();`



Why move semantics?

- `v = CreateAndInsert();`



Why move semantics?

- Totally, we call copy ctor for only 1 time, leading to only 1 copy.
- Besides, though we call dtor for 3 times, we don’t free any heap memory at all!
- 6 copy -> 1 copy
- 5 dtor -> 3 dtor
- 5 free -> no free

- Huge performance boost!

Value Category and Move Semantics

Move Ctor & Assignment

Move Ctor & Assignment

- Simplest example:

```cpp
#include <iostream>
class Object
{
public:
    Object() { std::cout << "Construct at " << this << "\n"; };
    ~Object() { std::cout << "Destruct at " << this << "\n"; };
    Object(const Object&) {
        std::cout << "Const Copy at " << this << "\n";
    };
    Object(Object&&) { std::cout << "Move at " << this << "\n"; };
    Object& operator=(const Object&) {
        std::cout << "Const Copy Assignment at " << this << "\n";
        return *this;
    };
    Object& operator=(Object&&) {
        std::cout << "Move Assignment at " << this << "\n";
        return *this;
    };
};
```

Move Ctor & Assignment

- It's not useful, just show how to define them;
  - Move semantics should "steal" another resource to itself.
- We might as well define a class called MyIntVector.

```cpp
class MyIntVector
{
    int *ptr_ = nullptr;
    std::size_t size_ = 0;

    void Clear_() noexcept { delete[] ptr_; }

    void Reset_() noexcept { ptr_ = nullptr, size_ = 0; }

    // This method assumes currently ptr_ = nullptr && size_ = 0.
    void AllocAndCopy_(const MyIntVector &another)
    {
        if (another.size_ == 0)
            return;
        std::unique_ptr<int[]> arr{ new int[another.size_] };
        std::memcpy(arr.get(), another.ptr_, another.size_ * sizeof(int));

        ptr_ = arr.release();
        size_ = another.size_;
        return;
    }
```

```cpp
public:
    MyIntVector(std::size_t initSize)
    {
        if (initSize == 0)
            return;
        ptr_ = new int[initSize], size_ = initSize;
    }
    int &operator[](std::size_t idx) { return ptr_[idx]; }
    int operator[](std::size_t idx) const { return ptr_[idx]; }
    auto size() const { return size_; }

    MyIntVector(const MyIntVector &another) { AllocAndCopy_(another); }
    ~MyIntVector() { Clear_(); }
    MyIntVector &operator=(const MyIntVector &another)
    {
        if (this == &another)
            return *this;

        Clear_();
        Reset_(); // For basic exception guarantee.
        AllocAndCopy_(another);
        return *this;
    }
```

Move Ctor & Assignment

- Move Ctor: just steal the resource!

```
MyIntVector(MyIntVector &&another)
{
    ptr_ = another.ptr_, size_ = another.size_;
    another.ptr_ = nullptr, another.size_ = 0;
}
```

```
MyIntVector(MyIntVector &&another)
    : ptr_{ std::exchange(another.ptr_, nullptr) },
      size_{ std::exchange(another.size_, 0) }
{
}
```

  - Utilize `std::exchange(x, y)`: assign y to x and return the original x.
- Move Assignment: stealing the resource & release self resource.

```
MyIntVector &operator=(MyIntVector &&another)
{
    if (this == &another)
        return *this;

    Clear_();
    ptr_ = std::exchange(another.ptr_, nullptr),
    size_ = std::exchange(another.size_, 0);
}
```

Move Ctor & Assignment

- There are still several questions that are worthwhile to discuss.
  - When will move ctor & assignment be called?
  - Why `const&` for copy but not `const&&` for move?
  - Is self-move legal?
  - How can I define move ctor & assignment for inheritance?
  - Should move ctor & assignment throw exceptions?

Move Ctor & Assignment

- 1. When will “move” instead of copy happen?
  - Still consider `std::string`; the only difference is the parameter: `const std::string&` or `std::string&&`?
    - The latter can only accept “rvalue”, and we call it “rvalue reference”.
    - rvalue can also be captured by `const&`, but with lower priority.
    - We’ll give an exact answer to them after teaching value category;
  - Roughly speaking, rvalues are:
    - Temporaries, like return value of the function;
    - Those **labeled** with `std::move`.
      - So `std::move` just labels a value as “rvalue” temporarily; the actual move operations are performed in move ctor & assignment!
      - For example, `std::move(a)` has no effect at all; `b = std::move(a)` moves `a` to `b`.

Move Ctor & Assignment

- For example:

Normal ctor, not copy or move.

Copy, since s is neither a temporary not labeled with std::move.

Move, since s + s returns a temporary.

Move, since there is a std::move.

Move, since CreateAndInsert() returns a temporary.

```
#include <vector>
#include <string>

std::vector<std::string> CreateAndInsert()
{
    std::vector<std::string> strVec;
    strVec.reserve(3); // prohibit reallocation to reduce complexity.
    std::string s = "data";

    strVec.push_back(s);
    strVec.push_back(s + s);
    strVec.push_back(std::move(s));

    return strVec;
}

int main()
{
    std::vector<std::string> v;
    v = CreateAndInsert();
    return 0;
}
```

Move Ctor & Assignment

- Consider: in `Func(std::string&& s)`, is `s` rvalue?
  - No, since it's neither temporary, nor labeled with `std::move`.
  - It just **refers to** a rvalue **from caller**, but in the callee it already becomes lvalue.

  - That's why someone may say "**rvalue reference is not necessarily rvalue**".
  - If you want to make it rvalue again, `std::move(s)`!

Move Ctor & Assignment

- Let’s see another example for move ctor.

```cpp
#include <string>

class Person
{
public:
    std::string name;
    int salary;

    Person(std::string init_name, int init_salary) :
        name{ init_name }, salary{ init_salary } {}

    Person(const Person& another) : name{ another.name },
        salary{ another.salary } {}

    Person(Person&& another) : name { std::move(another.name) },
        salary{ another.salary } {}

    Person& operator=(const Person& another) {
        name = another.name;
        salary = another.salary;
        return *this;
    }

    Person& operator=(Person&& another) {
        name = std::move(another.name);
        salary = another.salary;
        return *this;
    }
};
```

Naïve, we’ll improve it in the next lecture.

Move Ctor & Assignment

- Why isn't `salary` moved?
  - You can, but moving an integer is completely same as copying it.
  - For any fundamental type, moving is same as copying, and copying them is cheap enough.
  - But for a naïve `std::string`, it owns heap resources represented by a pointer, so copying it and then assigning it to `nullptr` steals it.
    - Much cheaper than copying heap memory!
- In fact, this can be generated automatically by compilers!
  - Implicit copy ctor -> member-wise copy ctor;
  - Implicit copy assignment -> member-wise copy assignment;
  - Implicit move ctor -> member-wise move ctor;
  - Implicit move assignment -> member-wise move assignment;

Move Ctor & Assignment

- Similar to `salary`, not every class needs move operations.
  - For example:
    ```
    struct A
    {
        int a, b;
    };
    ```
    - Nothing can be stolen.
  - Usually these classes can use the default-generated special member functions.
  - Exercise: does `std::array` have meaningful move operations?
    - Yes and no.
    - No: `std::array<T, N>` is `T[N]` on stack, so it cannot be stolen.
    - Yes: if `T` has meaningful move operations (`std::array<int>`❌, `std::array<std::unique_ptr<int>>`✅), then element-wise move is performed.
      - Compare: `std::vector` just exchanges the pointer (very cheap), while `std::array` needs member-wise move (cheaper than copy but not as cheap as `vector`).

`const&&`

- 2. Why is it `std::string&&` instead of `const std::string&&`?
  - Consider why we use `const std::string&` instead of `std::string&`.
    - We don’t want to modify anything in the parameter during a copy!
  - And consider what move ctor does:
    - It “steals” the members of the parameter, so modification is inevitable.
    - constness is not intended!
- Remember: `const&&` is almost always useless.
  - We’ll discuss it more in the next lecture.

Self-move

- 3. Is self-move legal?
  - i.e. `a = std::move(a);`
  - We've known that it just calls move assignment;
    - What if there isn't `if (&another == this) return *this;` in move assignment?

Oops! We release our own resource.

```
MyIntVector &operator=(MyIntVector &&another)
{


    Clear_();   void Clear_() noexcept { delete[] ptr_; }
    ptr_ = std::exchange(another.ptr_, nullptr),
    size_ = std::exchange(another.size_, 0);
}
```

- Copy ctor has similar problems if you don't check self assignment.
- All in all, it's still up to you whether self-move is legal.

Move Semantics Paradigm in Inheritance

- 4. How to define move ctor & assignment for inheritance?
  - It’s very similar to copy ctor & assignment in inheritance!
  - Base class:

```cpp
#include <vector>
#include <string>

class Person {
private:
    std::vector<std::string> tasks_;
public:
    Person(std::vector<std::string> initTasks) : tasks_{ std::move(initTasks) } {};
    Person(const Person& anotherPerson) : tasks_{ anotherPerson.tasks_ } {}
    Person(Person&& anotherPerson) : tasks_{ std::move(anotherPerson.tasks_) } {}
    Person& operator=(const Person& anotherPerson) {
        tasks_ = anotherPerson.tasks_; return *this;
    }
    Person& operator=(Person&& anotherPerson) {
        tasks_ = std::move(anotherPerson.tasks_); return *this;
    }
};
```

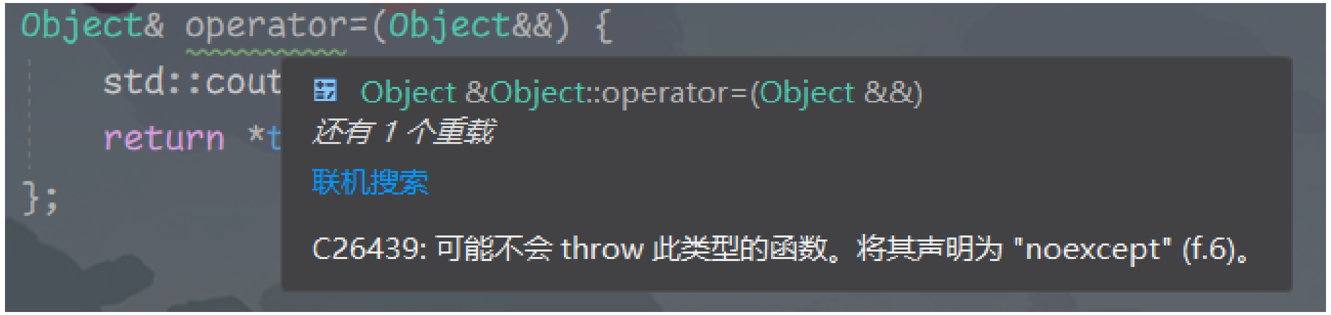
Move Semantics Paradigm in Inheritance

- Derived class:
  - *Maybe not good style because each line is too long.
  - Person{anotherStudent} involves implicit conversion from Student& to Person&.
    - Person{std::move(anotherStudent)}: Student&& to Person&&.
  - Person::operator= is to call the function of base class (otherwise = will call Student::operator= directly.)

```
class Student : public Person {
private:
    std::vector<std::string> evenMoreTasks_;
public:
    Student(std::vector<std::string> initTasks, std::vector<std::string> initEvenMoreTasks)
        : Person{ std::move(initTasks) }, evenMoreTasks_{ std::move(initEvenMoreTasks) } {};
    Student(const Student& anotherStudent) : Person{anotherStudent},
        evenMoreTasks_ { anotherStudent.evenMoreTasks_ } {}

    Student& operator=(const Student& anotherStudent) {
        evenMoreTasks_ = anotherStudent.evenMoreTasks_;
        Person::operator=(anotherStudent);
        return *this;
    }

};
```

noexcept in move semantics

- 5. Should move ctor & assignment throw exceptions?
  - We've said in *Error Handling* that you should use noexcept when you're sure no exception will be thrown.
  - In previous code pieces, you may notice that there is a green wave line.



  - But when you implement some other non-thrown functions (e.g. x * x), this warning will not be prompted.
  - This indicates that noexcept is so important for move ctor/assignment that it needs a special warning!

noexcept in move semantics

- Again, start by an example – the reallocation of std::vector.
  - Recap: when size == capacity, push_back() will trigger exponential reallocation to make it both dynamic and amortized $O(1)$ complexity.
  - The whole process is:
    - 1. Calculate the new capacity and allocate the new space;
    - 2. "Make" elements in the original space "go to" the new space;
    - 3. push back the new element to the new space;
    - 4. Deallocate the original space.
    - Notice that 2 and 3 can be reversed since we know where the new element should be.
  - And remember the exception safety guarantee of push_back?
    - Strong exception guarantee!
    - Meaning that if exception is thrown in push_back, the state is rolled back so the vector is still the original space with original elements.

Obvious move!

noexcept in move semantics

- So let’s try it…

```cpp
#include <string>
#include <iostream>
#include <vector>
#include <cassert>

class Person {
private:
    std::string name_;
public:
    Person(const char* n) : name_{ n } {}

    Person(const Person& p) : name_{ p.name_ } {
        std::cout << "COPY " << name_ << '\n';
    }
    Person(Person&& p) : name_{ std::move(p.name_) } {
        std::cout << "MOVE " << name_ << '\n';
    }
};
```

```cpp
int main()
{
    std::vector<Person> persons{ "Roach", "Ghost" };
    // To force any implementation to have capacity = 2.
    persons.shrink_to_fit();
    assert(persons.capacity() == 2);
    std::cout << "Test resizing...\n";
    // resize must happen.
    persons.push_back("Soap");
    return 0;
}
```

Output:
COPY Roach
COPY Ghost
/* Some possible shrinking related output,
depending on the platform… */
Test resizing…
MOVE Soap
COPY Roach
COPY Ghost

`noexcept` in move semantics

- The first two copies come from `std::initializer_list`
  - We've said that every element in `initializer_list` can be seen as `const`; thus the initialization is just copying element by element.
- When resizing, only the new element is moved, while the original elements are all copied!
  - Why? Is C++ so silly that this optimization is not enabled?
- Consider the exception safety guarantee, any idea?
  - If move may throw exception, how can you recover the original status?
    - Moving it back may cause another exception!
  - So, **to definitely succeed in rolling back**, `push_back` doesn't "steal" the original resources; when an exception is thrown, just destruct and deallocate new memory.

Output:
COPY Roach
COPY Ghost
Test resizing...
MOVE Soap
COPY Roach
COPY Ghost

noexcept in move semantics

- What if we use noexcept to guarantee move will not fail?

```
Person(Person&& p) noexcept : name_{ std::move(p.name_) } {
    std::cout << "MOVE " << name_ << '\n';
}
```

Output:
COPY Roach
COPY Ghost
Test resizing…
MOVE Soap
MOVE Roach
MOVE Ghost

- In most cases, move operations are:
  - If move assignment, destroy the original resource;
    - Similar to dtor.
  - "Steal" the resource of moved-from object.
  - We've said that dtor is assumed to be noexcept by the standard library; "steal" usually doesn't create new things so it's also Okay.
- Conclusion: move ctor/= should almost always be noexcept.

noexcept in move semantics

- You could use `std::is_nothrow_move_constructible_v<Type>` to test whether a type is move constructible with `noexcept`.
  - Assignment: `std::is_nothrow_move_assignible_v<Type>`.
- Brainstorming: what does this code piece mean?

```cpp
Person(Person&& p)
    noexcept(std::is_nothrow_move_constructible_v<std::string>
        && noexcept(std::cout << name_)
    ) : name_{ std::move(p.name_) }
{
    std::cout << "MOVE " << name_ << '\n';
}
```

除了直接使用 `noexcept` 外，函数还允许 `noexcept(bool)` 来定义noexcept限定。即

```cpp
void Func() noexcept(true); // same as void Func() noexcept;
void Func() noexcept(false); // same as void Func();
```

Our homework in *Error Handling*.

- Usually you don't need test `noexcept` like so, just FYI.

noexcept in move semantics

- Good news: if all data members will not throw exception when default constructed, default ctor (both implicit declaration and =default) will be noexcept by default.
  - Similarly, if all data members are nothrow_move_constructible, then the default move ctor is noexcept.
- So finally, a move ctor/assignment may be like:

```
MyIntVector(MyIntVector &&another) noexcept
    : ptr_{ std::exchange(another.ptr_, nullptr) },
      size_{ std::exchange(another.size_, 0) }
{
}
```

```
MyIntVector &operator=(MyIntVector &&another) noexcept
{
    if (this == &another)
        return *this;

    Clear_();
    ptr_ = std::exchange(another.ptr_, nullptr),
    size_ = std::exchange(another.size_, 0);
}
```

When to use `noexcept`

- Let's conclude when to use `noexcept` now.
  - For move ctor/assignment, `noexcept` or at least conditional `noexcept` is a must.
    - Reason: move is a very basic optimization; use copy because of no exception guarantee is costly.
  - Dtor should be `noexcept`, but it can be omitted since it's default.
  - Deallocation functions, including `operator delete`, should be `noexcept`.
  - Swap operations, like `.swap()` or friend function `swap()`.
    - Reason: some functions in the library may utilize it to optimize.
  - Those methods that **obviously** don't throw any exception.
    - Particularly, the hasher (like specialization of `std::hash`), `operator<=>/operator==` are better to be `noexcept`.
      - AFAIK libstdc++ has huge performance boost for unordered containers in such case.
      - This is also mentioned in Cpp Core Guideline [[C.85/86/89]]

Copy-and-swap idiom

- Wait, still one more thing…
  - Similar to copy assignment, we can also see obvious redundance in move assignment implementation.

```cpp
~MyIntVector() { Clear_(); }
MyIntVector(MyIntVector &&another) noexcept
    : ptr_{ std::exchange(another.ptr_, nullptr) }
    , size_{ std::exchange(another.size_, 0) }
{
}
```

```cpp
MyIntVector &operator=(MyIntVector &&another) noexcept
{
    if (this == &another)
        return *this;

    Clear_();
    ptr_ = std::exchange(another.ptr_, nullptr),
    size_ = std::exchange(another.size_, 0);
}
```

  - You can also utilize copy-and-swap idiom!

```cpp
void swap(MyIntVector &another) noexcept
{
    std::ranges::swap(ptr_, another.ptr_);
    std::ranges::swap(size_, another.size_);
}
```

```cpp
MyIntVector &operator=(MyIntVector &&another) noexcept
{
    if (this == &another)
        return *this;

    MyIntVector temp{ std::move(another) };
    swap(temp);
    return *this;
}
```

Copy-and-swap idiom

```cpp
// If implemented as member-wise swap.
MyIntVector &operator=(MyIntVector &&another) noexcept
{
    swap(another);
    return *this;
}
```

Again, another can be referred by MyIntVector& since it is lvalue.

- But unlike copy, copy-and-swap idiom in move assignment usually doesn't boost exception safety, **but just simplifies code**.
  - Since "steal" usually doesn't throw exceptions but copy may.
  - There are only few cases where copy-and-swap is a **must**, like for `std::shared_ptr`. We'll have a discussion in *Memory Management*.
- Similarly, copy-and-swap is slightly expensive than operating manually.

- Some may also suggest that move assignment can be implemented as member-wise swap.
  - Equivalent to deferring the release of the original resource.
    - Why? Think about it.
  - But some may also discourage it since usually users expect immediate resource releasing instead of swapping, like Scott Meyer.
  - We'll discuss it more in homework.

Value Category and Move Semantics

Rule of Zero / Five

Rule of Zero / Five

- We've said that in some cases move/copy ctor/assignment can be generated automatically by compilers.
  - So what's the actual rules?
  - Table means: user declaration of row forces `col` to be `table[row][col]`
    - Notice that `=default` and `=delete` are also declarations.

user declaration of \ forces	default constructor	copy constructor	copy assignment	move constructor	move assignment	destructor
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any constructor	undeclared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
copy constructor	undeclared	user declared	defaulted	undeclared (fallback enabled)	undeclared (fallback enabled)	defaulted
copy assignment	defaulted	defaulted	user declared	undeclared (fallback enabled)	undeclared (fallback enabled)	defaulted
move constructor	undeclared	deleted	deleted	user declared	undeclared (fallback disabled)	defaulted
move assignment	defaulted	deleted	deleted	undeclared (fallback disabled)	user declared	defaulted
destructor	defaulted	defaulted	defaulted	undeclared (fallback enabled)	undeclared (fallback enabled)	user declared

Credit: Howard Hinnant, for the table adopted from https://howardhinnant.github.io/bloomberg_2016.pdf

Implicit generation rules*

- Columns of move ctor/assignment match what we've learnt:
  - Copy Ctor/Assignment/Dtor/Another move function will disable implicit move declaration.
- Once users declare normal ctor explicitly, default ctor is disabled.
- Once users declare move ctor/assignment, default copy is disabled.
  - Why no dtor/another copy function?
  - C++ views this as design flaw and marks it as deprecated, so it's better to declare default copy explicitly.
  - So defining dtor only will astonishingly make move operations actually copy.

Rule of Zero / Five

- WTF, too many rules to remember.
  - It's also hard for maintainers to quickly understand whether a class is copyable/moveable.
- **Rule of Five**: Explicitly declare copy ctor & assignment, move ctor & assignment, dtor.
  - You can use `=default` and `=delete` if necessary, but declare explicitly!
  - Special case: what if all of them are `=default`?
    - That is, what if just member-wise copy/move/destruction?
- **Rule of Zero**: Don't explicitly declare **any of them** if possible.
  - Besides reducing code lines, there is bonus for rule of zero.
    - Trivially copyable!

BTW, in C++98 Rule of Five is Rule of Three, not useful in modern C++.

In *Lifetime* section.

Rule of Zero

Additionally, trivially copiable type is **exactly** the type that can be safely `std::memcpy`, `std::memmove` and `std::bit_cast`. Otherwise it's not safe to copy byte-wise.

- Recap:

We first need to introduce **trivial** special member functions.

- For normal ctor or move/copy ctor/assignment, if
  - It's implicitly declared or declared with `=default`.
  - All non-static data members and base classes have trivial one.
  - Class has no virtual base class and virtual member function.
- For dtor, as we've said, if
  - It's implicitly declared or `=default`.
  - It's non-virtual and all non-static data members have trivial dtor.

- Then trivially copyable means:
  - For move/copy ctor/assignment, at least one of them exist and all existing ones are trivial;
  - For dtor, it needs to be trivial.
  - Or if it's not a class, it can be scalar types (like integers), or an array of trivially copyable objects.
  - You can use `std::is_trivially_copyable_v<T>` to check it.

If you define them when it could be defaulted generated, sometimes it may lose the opportunity to become trivially copyable.

Trivially Copyable

- For example:

A is trivially-copyable but B isn't!

```
class A
{
public:
    int a;
};
static_assert(std::is_trivially_copyable_v<A>);

class B
{
public:
    B(const B &b) : a{ b.a } {}
    int a;
};
static_assert(std::is_trivially_copyable_v<B>);
```

Move-only class

- Particularly, `=delete` copy ctor & assignment while keeping move ones, class is uncopiable but only moveable.
  - Many examples in standard library: `std::unique_ptr`, `std::thread/std::jthread`, `std::fstream/stringstream/…`, `std::future/std::promise/…`
  - Usually it means holding some resource uniquely.
  - They have some common characteristics:
    - 1. Have APIs to check whether they actually own resources, e.g. `std::thread::joinable()`, conversion to `bool`, etc.
    - 2. Have APIs to release owned resources, e.g. `std::thread::join()`, `std::unique_ptr::reset()`, `std::fstream::close()`, etc.
    - 3. range-based for loop cannot use `auto/const auto`, because it involves copy.
      - For example: `std::vector<std::unique_ptr<int>> vec;`

```
for (auto ptr : vec)
{
}
```

Compile error ❌

Move-only class

- 4. cannot use `std::initializer_list<MoveOnlyClass>` to initialize `Container<MoveOnlyClass>`.
  - Reason: `std::initializer_list` provides a `const` view, so the container has to **copy** element by element, which is forbidden by move-only class.
  - Example:
    ```
    std::vector<std::unique_ptr<int>> vec{ new int{ 0 }, new int{ 1 } };
    ```
    Compile error ❌
  - Solution:
    ```
    std::vector<std::unique_ptr<int>> vec;
    vec.reserve(2);
    vec.emplace_back(new int{ 0 });
    vec.emplace_back(new int{ 1 });
    ```
  - Sooner we'll have some alternative solutions by "move iterators".

Value Category and Move Semantics

Moved-from states

Moved-from states

- Usually, a class has some invariants to maintain.
  - For example, for `MyString`, you may assume it has a valid pointer and has null termination.
  - But move operations may corrupt invariants.
    - E.g. you steals `ptr` and assigns it to `nullptr`.
    - And those methods that assume the invariants will also be invalid.
- So you should keep an eye on moved-from objects.
  - Generally, the least invariants for the class are:
    - Destructible without error.
    - Assignable without error.
      - And when it's assigned, it goes into a normal state again.
- Let's see some examples.

Typical Invariants and Violations

- Case1: some data members have certain formats.
- e.g. a Card class whose member is a string with "-of-", e.g. "queen-of-hearts".
- It also has a function called Print to print its rank and suit.
- Assuming this invariant, it can be coded as:
  - Then it has default move ctor & assignment, i.e. member-wise move.

```
class Card
{
private:
    std::string cardFullName_;
public:
    Card(const std::string& init_fullName) :cardFullName_(init_fullName)
    {
        if (init_fullName.find("-of-") == std::string::npos)
            throw std::invalid_argument{ "no -of- in the argument." };
    }

    void Print()
    {
        auto pos = cardFullName_.find("-of-");
        std::cout << std::format("{} {}\n",
            cardFullName_.substr(0, pos),
            cardFullName_.substr(pos + 4));
    }
};
```

Typical Invariants and Violations

- So when `Card` is moved, `cardFullName_` is empty.
  - What if we call `Print` now?
  - `substr` will return `std::string::npos` so `pos + 4` is 3, leading to out-of-range access.
- Case2: constraints between some data members.
  - For example, if move ctor/assignment of `MyIntVector` doesn't change size of moved-from objects.
  - Then some operations like `.size()` is meaningless!

Moved-from states

- Since the moved-from states are completely determined by the implementation of the class, there are several solutions.
  - 1. Make moved-from states satisfy current invariants.
    - For `MyIntVector`, `size_` should be assigned to 0.
    - For `Card`, make `cardFullName_` "-of-", but this makes moved objects to still own some resources.
      - But then the moved-from state of `Card` is not 100% natural and convincing, since they still own some resources.
      - `new` in move operations is likely to be discouraged by many.

Moved-from states

- 2. Enlarge the invariants, so that all member functions should consider moved-from states.
  - For `MyIntVector`, `nullptr` with `size == 0` is considered in every member function.
  - For `Card`, empty state is considered in every member function.
    - Like check `if (pos == std::string::npos)` in `Print`.
- 3. Explicitly document what is allowed for moved-from objects and what is prohibited.
  - You may use `assert` to check them.
- For users, pay attention to operations on moved-from objects before it’s assigned with a new state.

Value Category and Move Semantics

Algorithms and Iterators for Move Semantics

Algorithms for Move Semantics

- There are several kinds of algorithms that utilize move semantics.
- 1. Algorithms that directly represent move.
    - `std::move(inputBegin, inputEnd, outputBegin);`
        - Supposing `outputEnd = outputBegin + std::distance(inputBegin, inputEnd)`, this API moves element by element from `[inputBegin, inputEnd)` to `[outputBegin, outputEnd)` and returns `outputEnd.`
            - Just `std::copy` with move operations.
        - Equivalent to:

```
while (inputBegin != inputEnd)
{
    *outputBegin = std::move(*inputBegin);
    ++outputBegin, ++inputBegin;
}

return outputBegin;
```

        - Same name as `std::move(arg)`, overloading.

Algorithms for Move Semantics

- `std::move_backward(inputBegin, inputEnd, outputEnd);`
  - Similarly, `std::copy_backward` with move operations.
  - Equivalent to:

```
while (inputEnd != inputBegin)
{
    --inputEnd, --outputEnd;
    *outputEnd = std::move(*inputEnd);
}
*outputEnd = std::move(*inputEnd); // move the first one.
return outputEnd;
```

- E.g. Remember a possible implementation of `std::vector::insert`?
  - Reallocation if needed.
  - Then we need to shift elements after the inserted position…
    - Need a end-to-begin move to prevent overwriting!
  - This can use `std::move_backward` directly.
- `stdr::move/move_backward(inputRange, outputIt);`

Algorithms for Move Semantics

- 2. Algorithm for removing elements.
  - All functions in `<algorithm>` cannot really “delete” elements, but just return a range with non-removed elements.
    - The removed elements are in other parts with unspecified states, so you can use move semantics.
  - `std::remove(begin, end, value)`: remove all elements that are equal to `value`; return an iterator `mid` so that `[begin, mid)` contains all non-removed elements sequentially.
    - `std::erase` for containers.
  - `std::remove_if(begin, end, Pred)`: similarly, but the removing condition is not `elem == value` but `Pred(elem)`.
  - Implementation: dual pointers; one before means non-removed and the other after means scanning; then `*before == std::move(*after)` if the condition is satisfied.

Algorithms for Move Semantics

- `std::unique(begin, end)`: make all **adjacent** elements unique, and returns `mid` so that `[begin, mid)` is the uniquified sequence.
  - E.g. 1 1 2 3 3 3 1 => 1 2 3 1
  - Similarly, you can use dual pointer to solve this problem.
- `std::shift_left/right` in C++23.
- 3. Many other algorithms that may utilize move operations implicitly.
  - E.g. `std::rotate` by circular shift; `std::reverse` by swapping, which may use move semantics.

Iterators for Move Semantics

- We've said that using `std::initializer_list` to initialize containers will copy instead of move.
  - E.g.

```
std::vector<Person> persons{ "Roach", "Ghost" };
```

Output:
COPY Roach
COPY Ghost

- Usually, initial size is small so it's acceptable to copy.
  - However, what if it's either very expensive, or the elements are move-only?
- Besides pushing one by one manually, you can also use move iterators!

Iterators for Move Semantics

- Example:

```
Person initPersons[] = { "Roach", "Ghost" };
std::vector<Person> persons{
    std::move_iterator{ std::begin(initPersons) },
    std::move_iterator{ std::end(initPersons) }
};
```

Output:
MOVE Roach
MOVE Ghost

- This is obviously easier and cheaper than `persons.resize(2); stdr::move(initPersons, persons.begin());`
- Note: we use CTAD in C++17 to elide template parameter; before C++17, you can use `std::make_move_iterator(…)` like:

```
Person initPersons[] = { "Roach", "Ghost" };
std::vector<Person> persons{
    std::make_move_iterator(std::begin(initPersons)),
    std::make_move_iterator(std::end(initPersons))
};
```

Iterators for Move Semantics

- You can also use move iterators on one-pass algorithms, e.g. `std::for_each`:
    - One-pass is to guarantee moved objects are not used again, unless you know the moved-from state explicitly.

```
void process(std::string&&) {};

int main()
{
    std::vector<std::string> vec;

    std::for_each(
        std::move_iterator{ vec.begin() },
        std::move_iterator{ vec.end() },
        [](auto&& elem) { // or rvalue reference of the element.
            if (elem.empty())
                process(std::move(elem));
        }
    );
    return 0;
}
```

Summary

- Intro
  - Performance boost
  - Informal understanding of “rvalue”
  - `std::move`.
- Move ctor / assignment
  - Informal understanding of “rvalue reference”.
  - Useless const&&
  - Self-move
  - Exception-related.
    - noexcept, copy-and-swap idiom.
  - Inheritance-related.
- Rule of Zero/Five
  - Trivially copyable.
  - Move-only class
- Moved-from states
- Algorithms & iterators for move semantics

Next lecture…

- We’ll formally introduce value category.
  - So we can correctly understand what rvalue reference is.
  - And `decltype`!
  - We’ll then do some practices on analyzing cost by parameter choices.
  - And also cover some advanced topics, like return value optimization.