Back to Basics:
RAII and the Rule of Zero

Arthur O'Dwyer
2019-09-17

Outline

- Motivating the special members and the Rule of Three [3–24]
- Curious pitfall with (indirect) self-copy [25–37]
- RAII and exception safety [38–42]
- Deleting, defaulting, and the Rule of Zero [43–46]
- Move semantics and the Rule of Five (or Four) [47–54]
- Recap and examples of RAII types [55–64]
- Questions?

Classes that manage resources

A “resource,” for our purposes, is anything that requires special (manual) management.

C++ programs can manage many different kinds of “resources.”

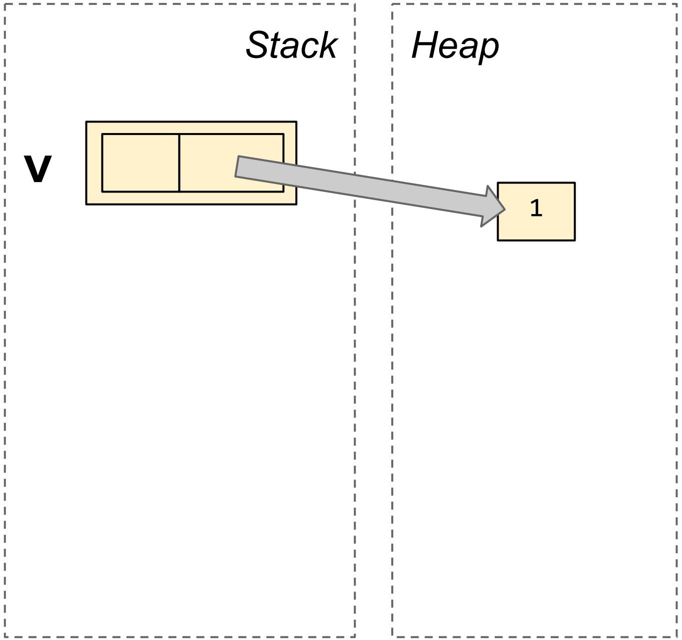
- Allocated memory (`malloc/free`, `new/delete`, `new[]/delete[]`)
- POSIX file handles (`open/close`)
- C FILE handles (`fopen/fclose`)
- Mutex locks (`pthread_mutex_lock/pthread_mutex_unlock`)
- C++ threads (spawn/`join`)
- Objective-C resource-counted objects (`retain/release`)

Classes that manage resources

Some of these resources are intrinsically “unique” (e.g. mutex locks), and some are “duplicable” (e.g. heap allocations; POSIX file handles can be `dup`’ed). For our purposes so far, this doesn’t really matter.

What matters is that there is some explicit action that needs to be taken by the program in order to ***free*** the resource.

We’ll stick with the classic boring example of heap allocation.



A naïve implementation of vector

```
class NaiveVector {
    int *ptr_;
    size_t size_;
public:
    NaiveVector() : ptr_(nullptr), size_(0) {}
    void push_back(int newvalue) {
        int *newptr = new int[size_ + 1];
        std::copy(ptr_, ptr_ + size_, newptr);
        delete [] ptr_;
        ptr_ = newptr;
        ptr_[size_++] = newvalue;
    }
};
```

This constructor correctly (if trivially) initializes `ptr_` with a resource.

This dance correctly replaces the resource managed by `ptr_`. No resource leaks here!

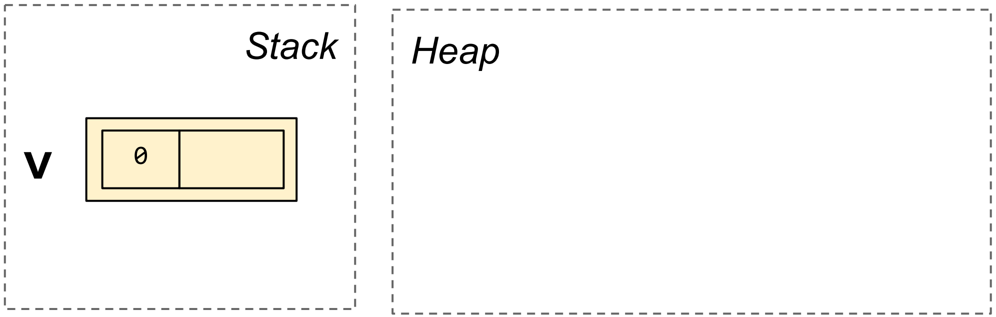
A naïve implementation of vector

```
{
    NaiveVector vec;  // here ptr_ is initialized with 0 elements
    vec.push_back(1); // ptr_ is correctly updated with 1 element
    vec.push_back(2); // ptr_ is correctly updated with 2 elements
}
```

So what's the problem?

A naïve implementation of vector

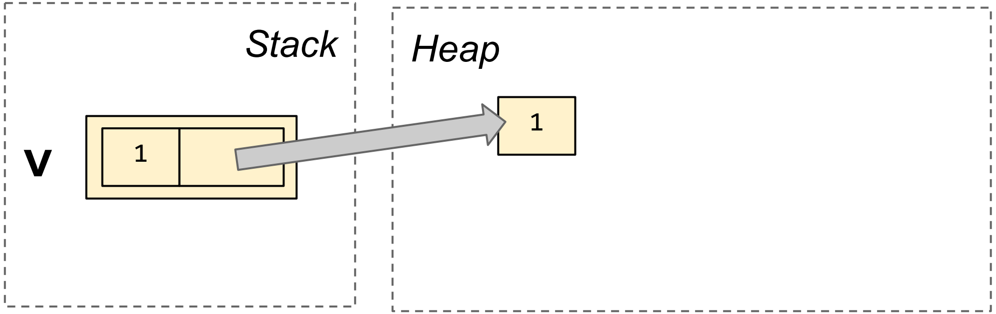
```
{
    NaiveVector vec;   // here ptr_ is initialized with 0 elements
    vec.push_back(1); // ptr_ is correctly updated with 1 element
    vec.push_back(2); // ptr_ is correctly updated with 2 elements
}
```



So what's the problem?

A naïve implementation of vector

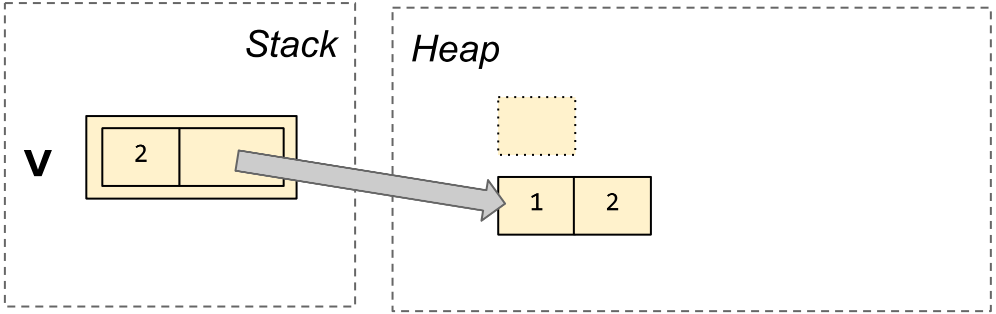
```
{
    NaiveVector vec;   // here ptr_ is initialized with 0 elements
    vec.push_back(1); // ptr_ is correctly updated with 1 element
    vec.push_back(2); // ptr_ is correctly updated with 2 elements
}
```



So what's the problem?

A naïve implementation of vector

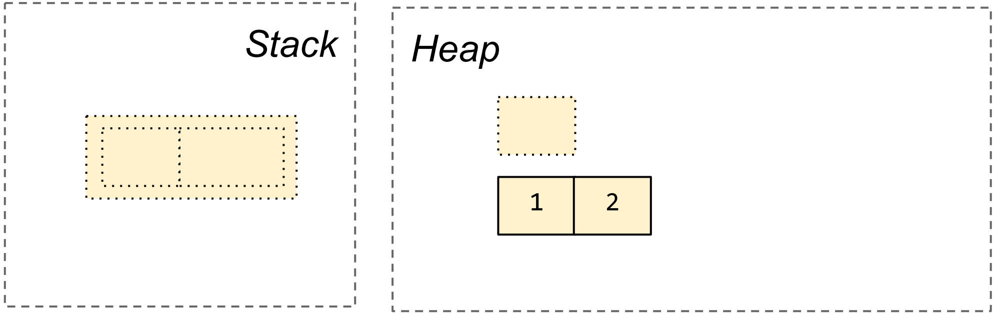
```
{
    NaiveVector vec;   // here ptr_ is initialized with 0 elements
    vec.push_back(1); // ptr_ is correctly updated with 1 element
    vec.push_back(2); // ptr_ is correctly updated with 2 elements
}
```



So what's the problem?

A naïve implementation of vector

```
{
    NaiveVector vec;   // here ptr_ is initialized with 0 elements
    vec.push_back(1); // ptr_ is correctly updated with 1 element
    vec.push_back(2); // ptr_ is correctly updated with 2 elements
}
```



Oops! Resource leak!

A naïve implementation of vector

```
class NaiveVector {
    int *ptr_;
    size_t size_;
public:
    NaiveVector() : ptr_(nullptr), size_(0) {}
    void push_back(int newvalue) {
        int *newptr = new int[size_ + 1];
        std::copy(ptr_, ptr_ + size_, newptr);
        delete [] ptr_;
        ptr_ = newptr;
        ptr_[size_++] = newvalue;
    }
};
```

The problem with this implementation of `vector` is that it leaks memory.

When the vector is in use, it allocates and deallocates its buffers correctly.

But when we're done with a vector object, we just drop the active pointer on the floor. We need to `delete[]` the active pointer when the vector object is destroyed.

Introducing the destructor

- When any object of class type is created, the compiler generates a call to a constructor of that type.
- Likewise, when any object's lifetime ends, the compiler generates a call to the ***destructor*** of that type.

```
{
    NaiveVector vec;    // here the constructor is called
}                       // here the destructor is called
```

Introducing the destructor

```
class NaiveVector {
    int *ptr_;
    size_t size_;
public:
    NaiveVector() : ptr_(nullptr), size_(0) {}
    void push_back(int newvalue) {
        int *newptr = new int[size_ + 1];
        std::copy(ptr_, ptr_ + size_, newptr);
        delete [] ptr_;
        ptr_ = newptr;
        size_ += 1;
    }
    ~NaiveVector() { delete [] ptr_; }
};
```

A destructor declaration looks just like a constructor declaration, but with a tilde ~ in front.

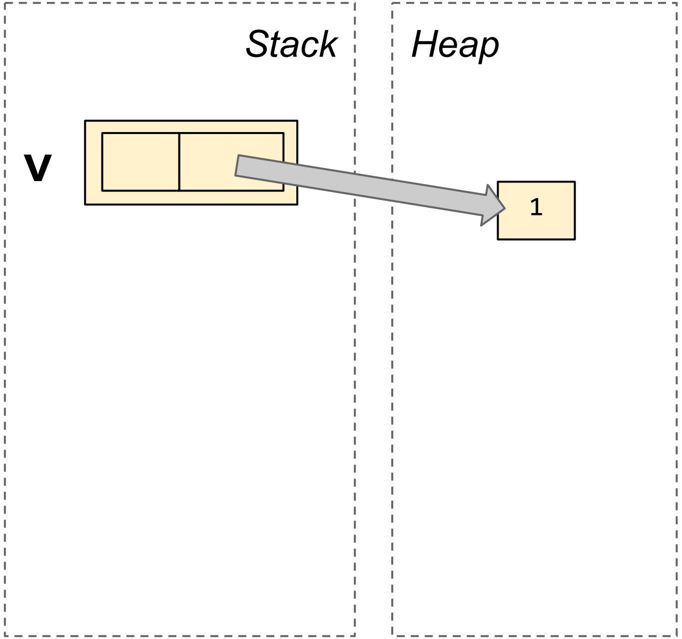
The destructor has no return type.

A class may have many overloaded constructors, but there is only ever a single destructor.

This `NaiveVector` no longer leaks memory on destruction.

NaiveVector still has bugs, though

```
{
  NaiveVector v;
  v.push_back(1);
  {
    NaiveVector w = v;
  }
  std::cout << v[0] << "\n";
}
```

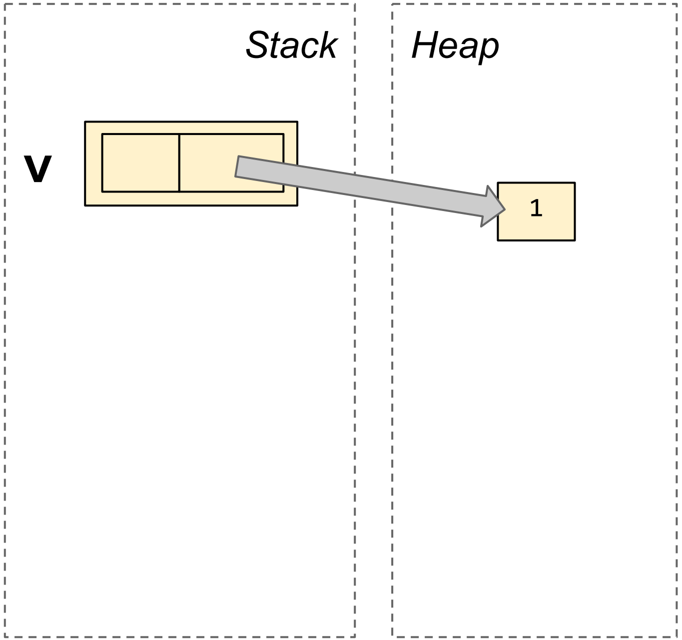


NaiveVector still has bugs, though

```
{
  NaiveVector v;
  v.push_back(1);
  {
    NaiveVector w = v;
  }
  std::cout << v[0] << "\n";
}
```

This line invokes the implicitly generated (defaulted) copy constructor of `NaiveVector`.

A defaulted copy constructor simply copies each member.

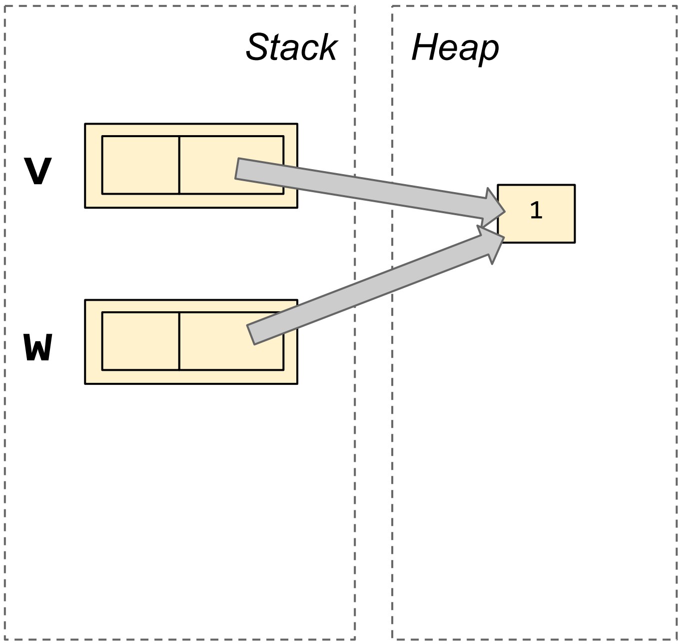


NaiveVector still has bugs, though

```
{
  NaiveVector v;
  v.push_back(1);
  {
    NaiveVector w = v;
  }
  std::cout << v[0] << "\n";
}
```

This line invokes the implicitly generated (defaulted) copy constructor of `NaiveVector`.

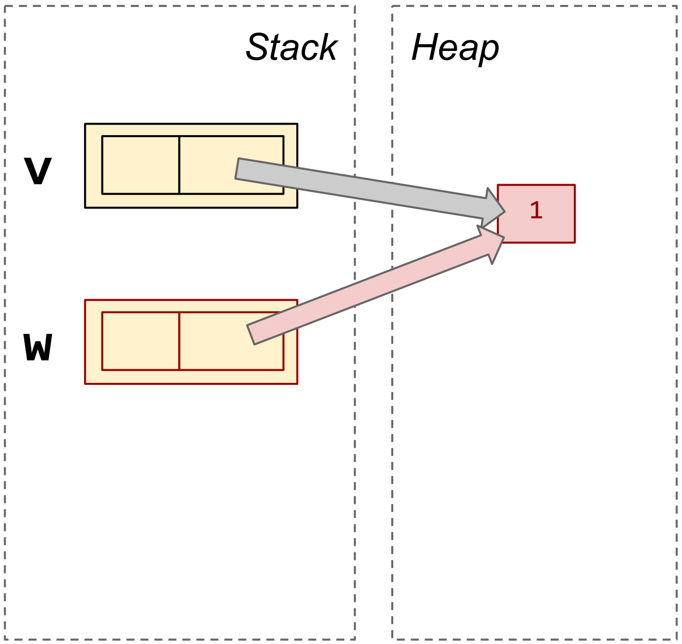
A defaulted copy constructor simply copies each member.



NaiveVector still has bugs, though

```
{
  NaiveVector v;
  v.push_back(1);
  {
    NaiveVector w = v;
  }
  std::cout << v[0] << "\n";
}
```

This line invokes the destructor we wrote for `NaiveVector`, which calls `delete [] ptr_`.

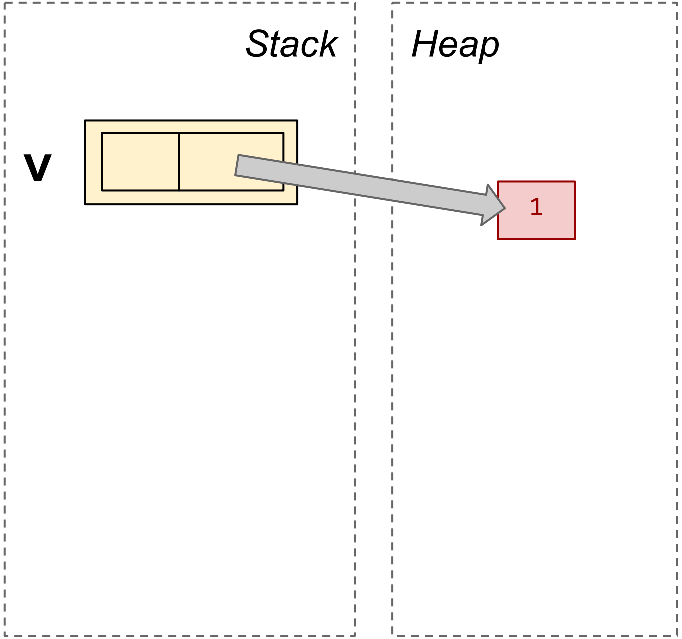


NaiveVector still has bugs, though

```
{
  NaiveVector v;
  v.push_back(1);
  {
    NaiveVector w = v;
  }
  std::cout << v[0] << "\n";
}
```

Now this line accesses memory that has already been freed.

Undefined behavior!

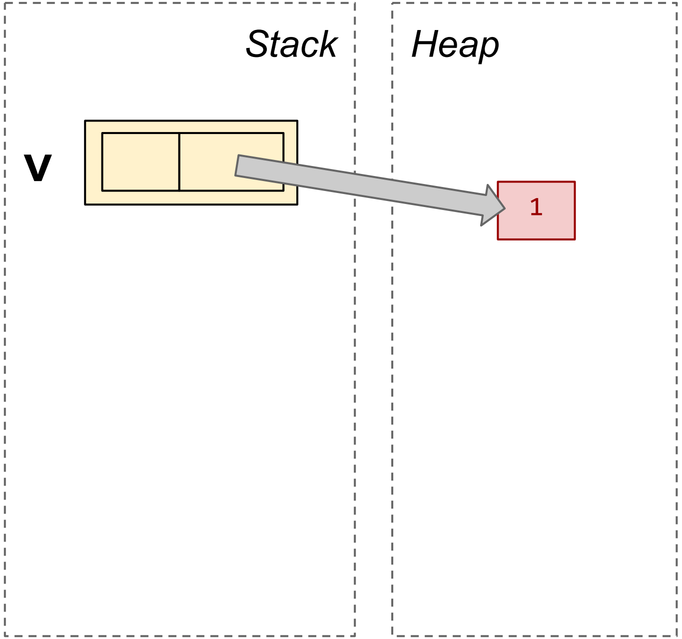


NaiveVector still has bugs, though

```
{
  NaiveVector v;
  v.push_back(1);
  {
    NaiveVector w = v;
  }
  std::cout << v[0] << "\n";
}
```

Finally, this line invokes the destructor of v, which calls `delete [] ptr_` ***again***.

This kind of bug is called a "double delete" or "double free."



Introducing the copy constructor

```
class NaiveVector {
    int *ptr_;
    size_t size_;
public:
    NaiveVector() : ptr_(nullptr), size_(0) {}
    ~NaiveVector() { delete [] ptr_; }

    NaiveVector(const NaiveVector& rhs) {
        ptr_ = new int[rhs.size_];
        size_ = rhs.size_;
        std::copy(rhs.ptr_, rhs.ptr_ + size_,
                  ptr_);
    }
};
```

Whenever you write a destructor, you probably need to write a copy constructor as well.

The destructor is responsible for freeing resources to avoid ***leaks***. The copy constructor is responsible for duplicating resources to avoid ***double frees***.

This applies to memory, or any other resource you might be managing.

Initialization is not assignment

Don’t confuse the = used for ***initialization*** with ***assignment!***

```
NaiveVector w = v;
```

This is an initialization (**construction**) of a **new** object.
It calls a copy constructor.

```
NaiveVector w;
w = v;
```

This is an **assignment** to the **existing** object `w`.
It calls an assignment operator.

Assignment has the same problem

```
{
  NaiveVector v;
  v.push_back(1);
  {
    NaiveVector w;
    w = v;
  }
  std::cout << v[0] << "\n";
}
```

This line invokes the implicitly generated (defaulted) `operator=` of `NaiveVector`.

A defaulted copy assignment operator simply copy-assigns each member.

Introducing copy assignment

```
class NaiveVector {
    int *ptr_;
    size_t size_;
public:
    NaiveVector() : ptr_(nullptr), size_(0) {}
    ~NaiveVector() { delete [] ptr_; }
    NaiveVector(const NaiveVector& rhs) { ... }

    NaiveVector& operator=(const NaiveVector& rhs) {
        NaiveVector copy = rhs;
        copy.swap(*this);
        return *this;
    }
};
```

Whenever you write a destructor, you probably need to write a copy constructor ***and a copy assignment operator***.

This demonstrates the ***copy and swap*** idiom.

We need to write `swap`.

The Rule of Three

- If your class directly manages some kind of resource (such as a `new`'ed pointer), then you almost certainly need to hand-write three special member functions:
  - A ***destructor*** to free the resource
  - A ***copy constructor*** to copy the resource
  - A ***copy assignment operator*** to free the left-hand resource and copy the right-hand one
- Use the copy-and-swap idiom to implement assignment.

Why copy and swap?

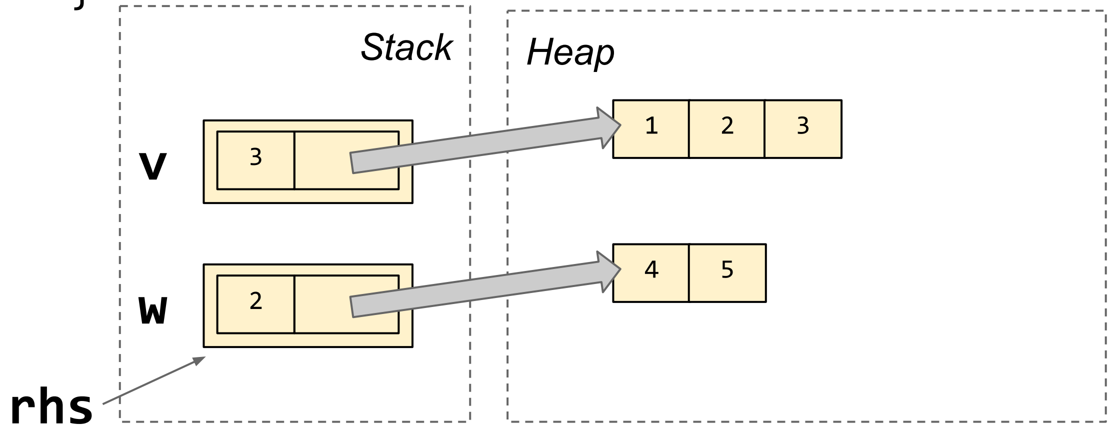
You might simply overwrite each member one at a time, like this.

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    delete ptr_;
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
    return *this;
}
```

But this code is not robust against ***self-assignment***.

Non-self-copy example

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    delete ptr_;
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
    return *this;
}
```

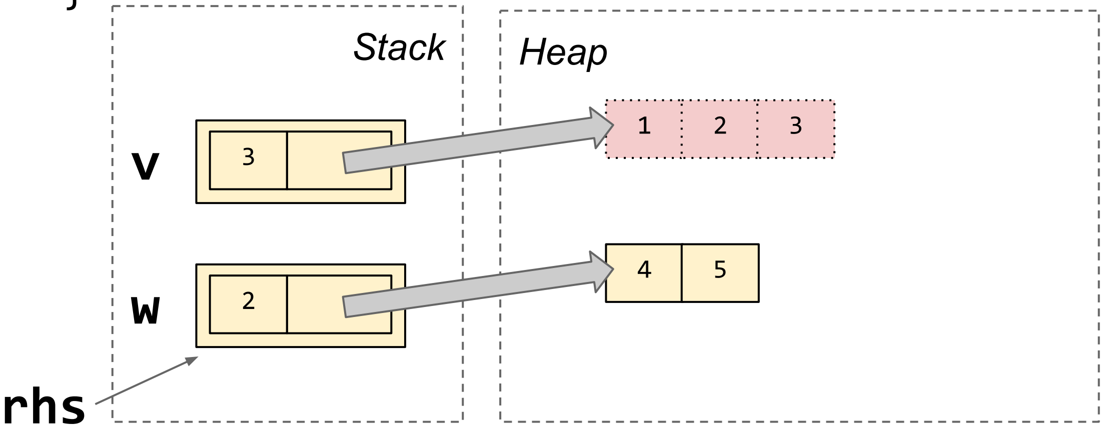


Not troublesome:

```
v = w;
```

Non-self-copy example

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    delete ptr_;
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
    return *this;
}
```

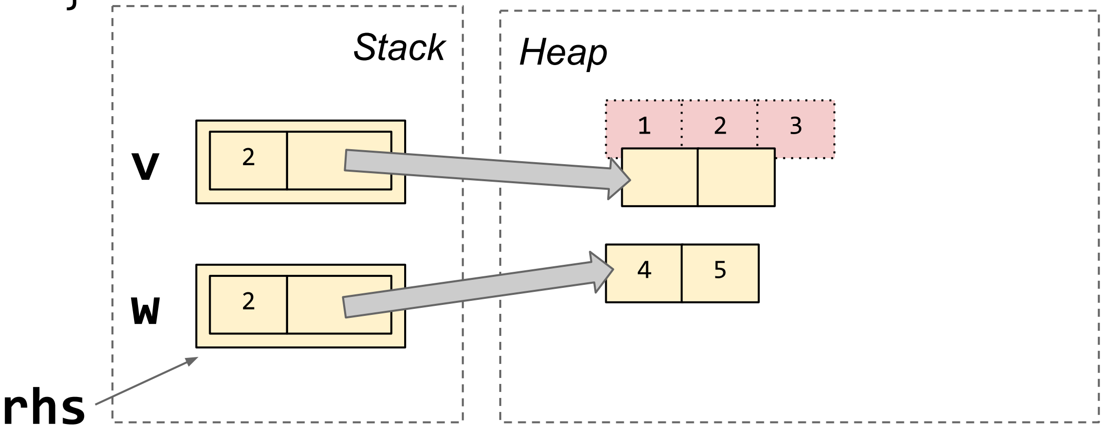


Not troublesome:

```
v = w;
```

Non-self-copy example

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    delete ptr_;
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
    return *this;
}
```

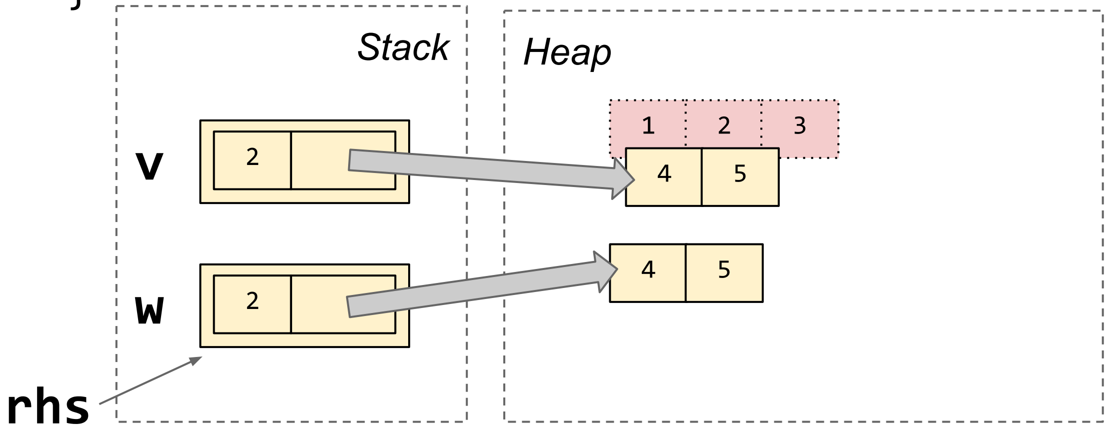


Not troublesome:

```
v = w;
```

Non-self-copy example

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    delete ptr_;
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
    return *this;
}
```

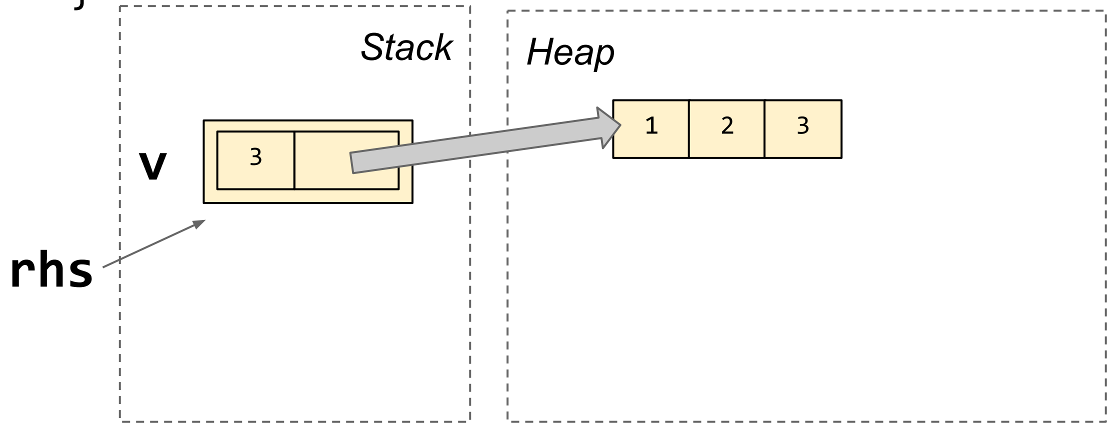


Not troublesome:

```
v = w;
```

Self-copy example 2 (problem)

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    delete ptr_;
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
    return *this;
}
```

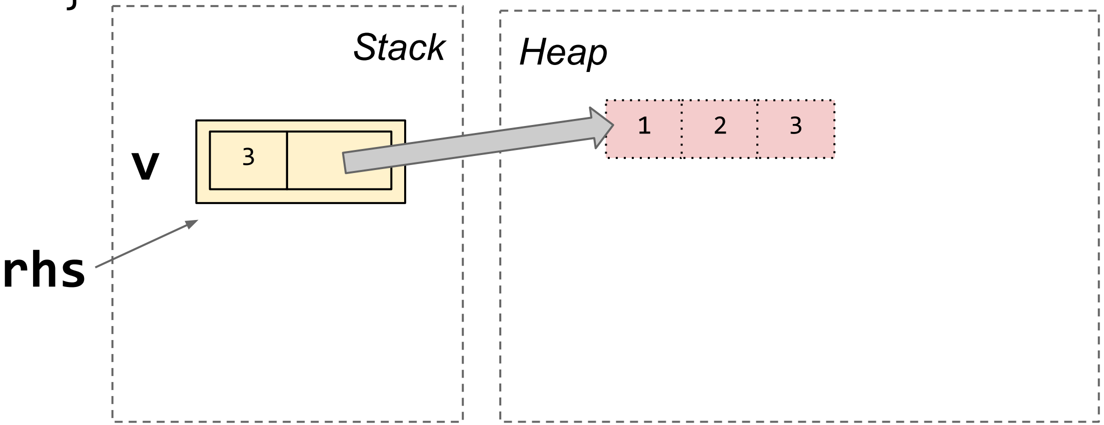


Troublesome:

`v = v;`

Self-copy example 2 (problem)

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    delete ptr_;
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
    return *this;
}
```

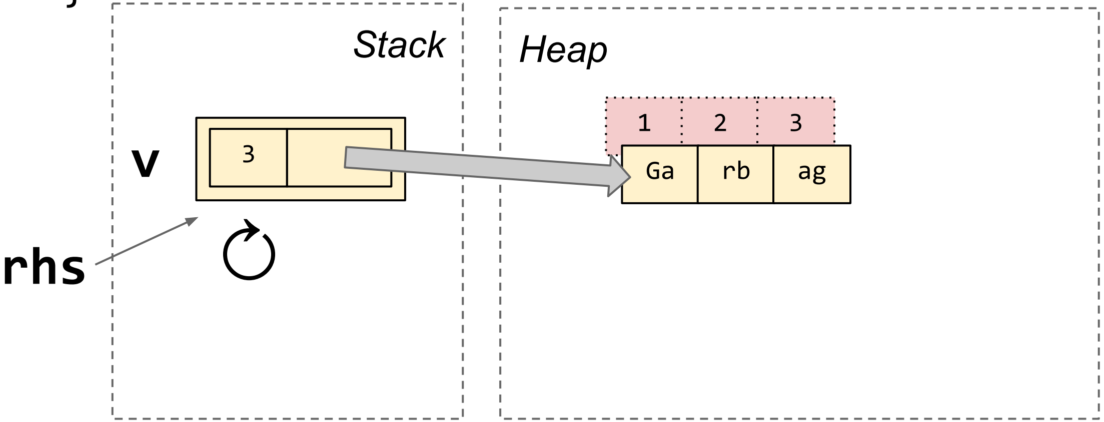


Troublesome:

```
v = v;
```

Self-copy example 2 (problem)

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    delete ptr_;
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
    return *this;
}
```

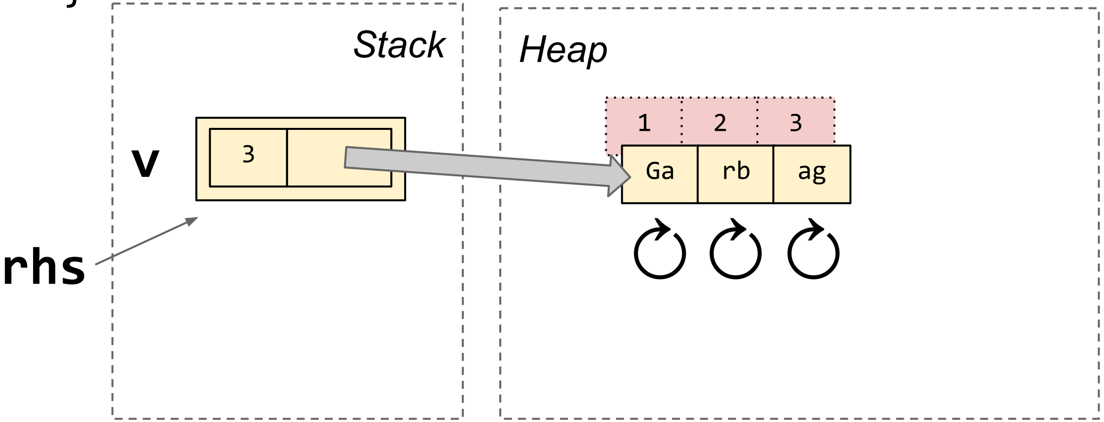


Troublesome:

```
v = v;
```

Self-copy example 2 (problem)

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    delete ptr_;
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
    return *this;
}
```



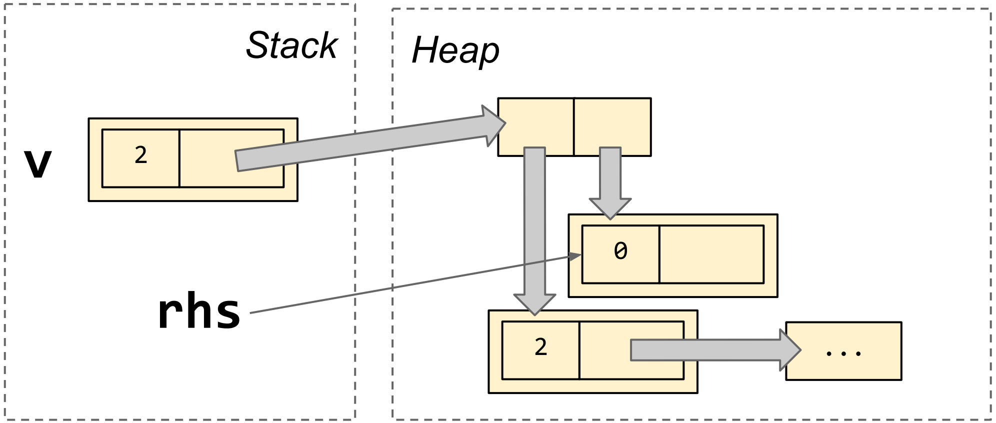
Troublesome:

```
v = v;
```

Thanks to Daniel Frey for this example.

Self-copy example 3 (problem)

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    delete ptr_;
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
    return *this;
}
```



Not self-move but still troublesome (for templated or recursive data structures):

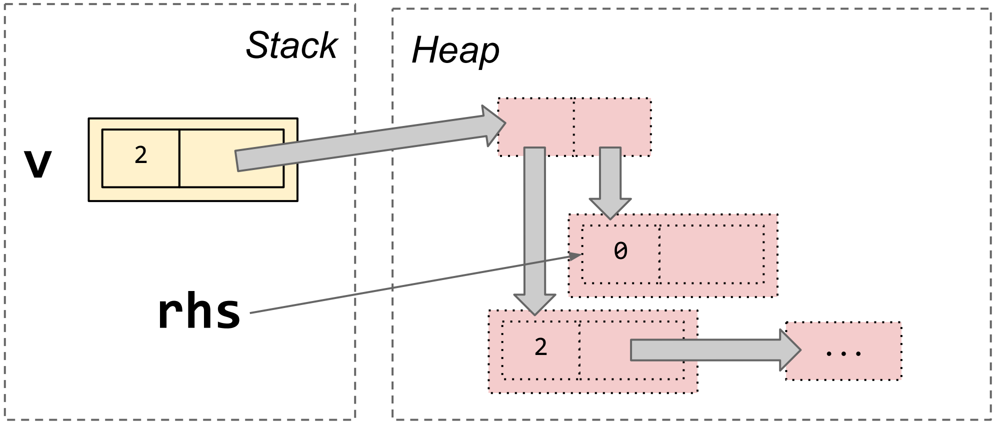
```
struct A {
  NaiveVector<shared_ptr<A>> m;
};

NaiveVector<shared_ptr<A>> v;
v = v[1]->m;
```

Thanks to Daniel Frey for this example.

Self-copy example 3 (problem)

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    delete ptr_;
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
    return *this;
}
```



Not self-move but still troublesome (for templated or recursive data structures):

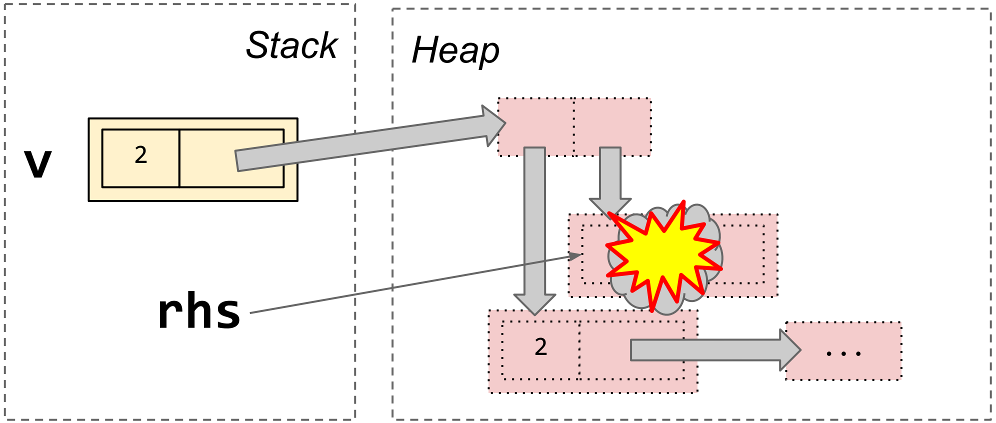
```
struct A {
  NaiveVector<shared_ptr<A>> m;
};

NaiveVector<shared_ptr<A>> v;
v = v[1]->m;
```

Thanks to Daniel Frey for this example.

Self-copy example 3 (problem)

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    delete ptr_;
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
    return *this;
}
```



Not self-move but still troublesome (for templated or recursive data structures):

```
struct A {
  NaiveVector<shared_ptr<A>> m;
};

NaiveVector<shared_ptr<A>> v;
v = v[1]->m;
```

Copy-and-swap to the rescue!

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    NaiveVector copy(rhs);
    copy.swap(*this);
    return *this;
}
```

We make a complete copy of `rhs` before the first modification to `*this`.

So any aliasing relationship between `rhs` and `*this` cannot trip us up.

RAII and exception safety

“Resource Acquisition Is Initialization.”

The slogan is about initialization,
but its meaning is really about ***cleanup***.

RAII and exception safety

Destructors help us write code that is robust against exceptions

- C++ supports `try/catch` and `throw`
- When an exception is thrown, the runtime looks “up the call stack” until it finds a suitable `catch` handler for the type of the exception being thrown. Assuming it finds one...
- The runtime performs ***stack unwinding***. For every local scope between the throw and the catch handler, the runtime invokes the destructors of all local variables in that scope.
- To avoid leaks, place all your ***cleanup*** code in ***destructors***.

RAII and exception safety

```
int main() {
  try {
    int *arr = new int[4];
    throw std::runtime_error("for example");
    delete [] arr;  // cleanup
  } catch (const std::exception& ex) {
    std::cout << "Caught an exception: " << ex.what() << "\n";
  }
}
```

RAII and exception safety

This code calls `new`, but fails to call `delete` when an exception is thrown. Therefore it leaks memory. This is not good RAII code.

```
int main() {
  try {
    int *arr = new int[4];
    throw std::runtime_error("for example");
    delete [] arr;  // cleanup
  } catch (const std::exception& ex) {
    std::cout << "Caught an exception: " << ex.what() << "\n";
  }
}
```

Oops!

RAII and exception safety

```
struct RAIIPtr {
    int *ptr_;
    RAIIPtr(int *p) : ptr_(p) {}
    ~RAIIPtr() { delete [] ptr_; }
};

int main() {
  try {
    RAIIPtr arr = new int[4];
    throw std::runtime_error("for example");
  } catch (const std::exception& ex) {
    std::cout << "Caught an exception: " << ex.what() << "\n";
  }
}
```

This code cannot fail to call `delete` even when an exception is thrown, because it places the call to `delete` in a destructor.

This is still relatively dangerous code because `RAIIPtr` has a defaulted copy constructor.

Deleted special member functions

```
struct RAIIPtr {
    int *ptr_;
    RAIIPtr(int *p) : ptr_(p) {}
    ~RAIIPtr() { delete [] ptr_; }

    RAIIPtr(const RAIIPtr&) = delete;
    RAIIPtr& operator=(const RAIIPtr&) = delete;
};
```

We can improve our `RAIIPtr` by making it ***non-copyable***.

When a function definition has the body `=delete;` instead of a curly-braced compound statement, the compiler will reject calls to that function at compile time.

This facility is completely unrelated to `new/delete`; it's just a cutesy use of an existing keyword. New keywords are expensive, because C++ values backward compatibility.

Defaulted special member functions

```
class Book {
    // ...

public:
    Book(const Book&) = default;
    Book& operator=(const Book&) = default;
    ~Book() = default;
};
```

When a special member function definition has the body `=default;` instead of a curly-braced compound statement, the compiler will create a defaulted version of that function, just as if it were implicitly generated.

Explicitly defaulting your special members can help your code to be self-documenting.

The Rule of Zero

- If your class does not *directly* manage any resource, but merely uses library components such as vector and string, then you should strive to write ***no*** special member functions.
  Default them all!
  - Let the compiler implicitly generate a defaulted destructor
  - Let the compiler generate the copy constructor
  - Let the compiler generate the copy assignment operator
  - (But your own `swap` might improve performance)
- This is known as the ***Rule of Zero***

Prefer Rule of Zero when possible

There are two kinds of well-designed value-semantic C++ classes:

- ***Business-logic classes*** that do not manually manage any resources, and follow the Rule of Zero
  - They *delegate* the job of resource management to data members of types such as `std::string`
- ***Resource-management classes*** (small, single-purpose) that follow the Rule of Three
  - Acquire the resource in each constructor; free the resource in your destructor; copy-and-swap in your assignment operator

Introducing rvalue references

- C++11 introduces ***rvalue reference*** types.
- The references we've seen so far are ***lvalue*** references.

The terms "lvalue" and "rvalue" come from the syntax of assignment expressions. An lvalue can appear on the left-hand side of an assignment; an rvalue must appear on the right-hand side.

```
int x, *p, a[10];
```

```
x = 1;
*p = 1;
a[2] = 1;
```

```
42 = 1;
&x = 1;
x+1 = 1;
```

x, *p, and a[2] are ***lvalues***.

42, &x, and x+1 are ***rvalues***.

Introducing rvalue references

- `int&` is an ***lvalue reference*** to an `int`.
- `int&&` (two ampersands) is an ***rvalue reference*** to an `int`.
- As a general rule, lvalue reference parameters do not bind to rvalues, and rvalue reference parameters do not bind to lvalues.
- Special case for backward compatibility: a `const` lvalue reference will happily bind to an rvalue.

```
void f(int&);         f(i); // OK     f(42); // ERROR
void g(int&&);        g(i); // ERROR  g(42); // OK
void h(const int&);   h(i); // OK     h(42); // OK!
```

Rvalues won’t be missed

Combine this with overload resolution...

```
void foo(const std::string&);  // takes lvalues
void foo(std::string&&);       // takes rvalues

std::string s = "hello";
foo(s);                    // calls foo #1
foo(s + " world");         // calls foo #2
foo("hi");                 // calls foo #2
foo(std::move(s));         // calls foo #2
```

Inside the body of `foo` #2 we can steal the guts of the string parameter, because it is a temporary (or has been `std::move`d by our caller).

Rvalues won’t be missed

The most common application of rvalue references is the ***move constructor***.

```
class NaiveVector {
    NaiveVector(const NaiveVector& rhs) {
        ptr_ = new int[rhs.size_];
        size_ = rhs.size_;
        std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
    }
    NaiveVector(NaiveVector&& rhs) {
        ptr_ = std::exchange(rhs.ptr_, nullptr);
        size_ = std::exchange(rhs.size_, 0);
    }
};
```

`new int` is slow.

`std::copy` is slow.

The move constructor doesn’t need to do either of those slow things!

Each STL container type has a move constructor in addition to its copy constructor.

The Rule of Five

- If your class directly manages some kind of resource (such as a `new`'ed pointer), then you may need to hand-write ***five*** special member functions for correctness *and* performance:
  - A ***destructor*** to free the resource
  - A ***copy constructor*** to copy the resource
  - A ***move constructor*** to transfer ownership of the resource
  - A ***copy assignment operator*** to free the left-hand resource and copy the right-hand one
  - A ***move assignment operator*** to free the left-hand resource and transfer ownership of the right-hand one

Copy-and-swap leads to duplication

Rather than write these two assignment operators,
whose code is almost identical...

```
NaiveVector& NaiveVector::operator=(const NaiveVector& rhs) {
    NaiveVector copy(rhs);
    copy.swap(*this);
    return *this;
}

NaiveVector& NaiveVector::operator=(NaiveVector&& rhs) {
    NaiveVector copy(std::move(rhs));
    copy.swap(*this);
    return *this;
}
```

By-value assignment operator?

...What if we just wrote one assignment operator and left the copy up to our caller?

I’m not aware of any problems with this idiom. However, it is relatively uncommon; writing copy assignment and move assignment separately is more frequently seen. In particular, the STL always writes them separately.

```
NaiveVector& NaiveVector::operator=(NaiveVector copy) {
    copy.swap(*this);
    return *this;
}
```

The Rule of Four (and a half)

- If your class directly manages some kind of resource (such as a `new`'ed pointer), then you may need to hand-write ***four*** special member functions for correctness *and* performance:
  - A ***destructor*** to free the resource
  - A ***copy constructor*** to copy the resource
  - A ***move constructor*** to transfer ownership of the resource
  - A ***by-value assignment operator*** to free the left-hand resource and transfer ownership of the right-hand one

  ½ (A nonmember ***swap*** function, and ideally a member version too)

No longer naïve vector

```
class Vec {
  Vec(const Vec& rhs) {
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
  }

  Vec(Vec&& rhs) noexcept {
    ptr_ = std::exchange(rhs.ptr_, nullptr);
    size_ = std::exchange(rhs.size_, 0);
  }

  friend void swap(Vec& a, Vec& b) noexcept {
    a.swap(b);
  }

  ~Vec() {
    delete [] ptr_;
  }

  Vec& operator=(Vec copy) {
    copy.swap(*this);
    return *this;
  }

  void swap(Vec& rhs) noexcept {
    using std::swap;
    swap(ptr_, rhs.ptr_);
    swap(size_, rhs.size_);
  }
};
```

No longer naïve vector

A destructor, to free the resource (avoid leaks)

```
class Vec {
  Vec(const Vec& rhs) {
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
  }

  Vec(Vec&& rhs) noexcept {
    ptr_ = std::exchange(rhs.ptr_, nullptr);
    size_ = std::exchange(rhs.size_, 0);
  }

  friend void swap(Vec& a, Vec& b) noexcept {
    a.swap(b);
  }
```

A copy constructor, to copy the resource (avoid double-frees)

A move constructor, to transfer ownership of the resource (cheaper than copying)

A two-argument swap, to make your type efficiently "std::swappable"

```
  ~Vec() {
    delete [] ptr_;
  }

  Vec& operator=(Vec copy) {
    copy.swap(*this);
    return *this;
  }

  void swap(Vec& rhs) noexcept {
    using std::swap;
    swap(ptr_, rhs.ptr_);
    swap(size_, rhs.size_);
  }
};
```

An assignment operator, to free the left-hand resource and transfer ownership of the right-hand one

A member swap too, for simplicity

No longer naïve vector

```cpp
class Vec {
  Vec(const Vec& rhs) {
    ptr_ = new int[rhs.size_];
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
  }

  Vec(Vec&& rhs) noexcept {
    ptr_ = std::exchange(rhs.ptr_, nullptr);
    size_ = std::exchange(rhs.size_, 0);
  }

  friend void swap(Vec& a, Vec& b) noexcept {
    a.swap(b);
  }

  ~Vec() {
    delete [] ptr_;
  }

  Vec& operator=(Vec copy) {
    copy.swap(*this);
    return *this;
  }

  void swap(Vec& rhs) noexcept {
    using std::swap;
    swap(ptr_, rhs.ptr_);
    swap(size_, rhs.size_);
  }
};
```

copy the resource

transfer ownership

free the resource

swap ownership

Closer-to-Rule-of-Zero vector

```
class Vec {
  std::unique_ptr<int[]> uptr_;
  int size_;
```

copy the resource

```
  Vec(const Vec& rhs) {
    uptr_ = std::make_unique<int[]>(rhs.size_);
    size_ = rhs.size_;
    std::copy(rhs.ptr_, rhs.ptr_ + size_, ptr_);
  }
```

transfer ownership

```
  Vec(Vec&& rhs) noexcept = default;

  friend void swap(Vec& a, Vec& b) noexcept {
    a.swap(b);
  }
```

free the resource

```
  ~Vec() = default;
```

free *and* transfer ownership

```
  Vec& operator=(Vec copy) {
    copy.swap(*this);
    return *this;
  }
```

swap ownership

```
  void swap(Vec& rhs) noexcept {
    using std::swap;
    swap(uptr_, rhs.uptr_);
    swap(size_, rhs.size_);
  }
};
```

True Rule-of-Zero vector

```
class Vec {
  std::vector<int> vec_;

  Vec(const Vec& rhs) = default;
  Vec(Vec&& rhs) noexcept = default;
  Vec& operator=(const Vec& rhs) = default;
  Vec& operator=(Vec&& rhs) = default;
  ~Vec() = default;

  void swap(Vec& rhs) noexcept {
    vec_.swap(rhs.vec_);
  }

  friend void swap(Vec& a, Vec& b) noexcept {
    a.swap(b);
  }
};
```

Memberwise assignment can hit Daniel's pitfall from slides 34–36. Use copy-and-swap if this is a concern for you.

swap ownership: now only for performance, not correctness

Examples of resource management

`unique_ptr` manages a raw pointer to a uniquely owned heap allocation.

- ***Destructor*** frees the resource
  - Calls `delete` on the raw pointer
- ***Copy constructor*** copies the resource
  - Copying doesn't make sense. We `=delete` this member function.
- ***Move constructor*** transfers ownership of the resource
  - Transfers the raw pointer, then nulls out the right-hand side
- ***Copy assignment operator*** frees the left-hand resource and copies the right-hand one
  - Copying doesn't make sense. We `=delete` this member function.
- ***Move assignment operator*** frees the left-hand resource and transfers ownership of the right-hand one
  - Calls `delete` on the left-hand ptr, transfers, then nulls out the right-hand ptr

Examples of resource management

`shared_ptr` manages a reference count.

- ***Destructor*** frees the resource
  - Decrements the refcount (and maybe cleans up if the refcount is now zero)
- ***Copy constructor*** copies the resource
  - Increments the refcount
- ***Move constructor*** transfers ownership of the resource
  - Leaves the refcount the same, then disengages the right-hand side
- ***Copy assignment operator*** frees the left-hand resource and copies the right-hand one
  - Decrements the old refcount, increments the new refcount
- ***Move assignment operator*** frees the left-hand resource and transfers ownership of the right-hand one
  - Decrements the old refcount, then disengages the right-hand side

Examples of resource management

`unique_lock` manages a lock on a mutex.

- ***Destructor*** frees the resource
  - Unlocks the mutex if “engaged”
- ***Copy constructor*** copies the resource
  - Copying doesn’t make sense. We `=delete` this member function.
- ***Move constructor*** transfers ownership of the resource
  - Leaves the mutex in the same state, then disengages the right-hand side
- ***Copy assignment operator*** frees the left-hand resource and copies the right-hand one
  - Copying doesn’t make sense. We `=delete` this member function.
- ***Move assignment operator*** frees the left-hand resource and transfers ownership of the right-hand one
  - Unlocks the old mutex if “engaged”; then transfers ownership from the right-hand side

Examples of resource management

`ifstream` manages a POSIX file handle and an associated buffer.

- ***Destructor*** frees the resource
  - Calls `close` on the handle
- ***Copy constructor*** copies the resource
  - We `=delete` this member function; but you could imagine calling `dup` on the handle (and giving the copy a fresh buffer, to avoid duplicated output)
- ***Move constructor*** transfers ownership of the resource
  - Transfers the handle and the contents of the buffer
- ***Copy assignment operator*** frees the left-hand resource and copies the right-hand
  - We `=delete` this member function; but you could imagine `close`/`dup`
- ***Move assignment operator*** frees the left-hand resource and transfers ownership of the right-hand one
  - Calls `close`, then transfers the handle and contents of the buffer

“Pilfering” implies an “empty” state

Each of the preceding examples had a move operation which involved “disengaging the right-hand side” or “nulling out the right-hand side.”

If you forget to do this, then you may have double-free bugs.

After your move operation pilfers the guts of the right-hand object without destroying it, the right-hand object must be left in a state “emptied of guts.”

You can do RAII with only copy and destroy operations, no move. In that case you have no empty state. But if making a copy is slow or impossible, then you won’t be able to go this route.

You can even do RAII with ***only*** destroy; just `=delete` your copy and move operations. `std::lock_guard` is an example.

Questions?