Introduction to Software Development – CS 6010
Lecture 16 – Operator Overloading

Master of Software Development (MSD) Program

Varun Shankar

Fall 2023

Destructor Syntax

- A Class Constructor is just the name of the class:
  - MyVec()::MyVec(int size);
- The Destructor is the class name with a ~ (tilde) in front:
  - MyVec()::~MyVec()

```
class MyVec {
        MyVec(); // Constructor
        ~MyVec(); // Destructor
}
```

- You almost never call a class destructor yourself! It is called automatically by the system (when the object is destroyed).

Destructors

- Where does the memory used to store all the numbers in a vector get allocated?
  - The Heap
- When we are done with a vector variable, do we ever delete that memory? How come the memory isn’t “leaked”?
- The vector automatically deletes any memory it used when it goes out of scope.
- This happens in a special function that is called when the vector is destroyed (goes out of scope / is explicitly destroyed).
- This function is called the *destructor*.
- It contains “cleanup code” that an object needs to cleanup after itself. This could include deleting heap memory, closing a file, etc.
- An object is destroyed when:
  - The programmer explicitly calls delete to deallocate an object on the heap.
  - A function returns and its stack frame is deallocated (goes out of scope).

Lecture 16 – Operator Overloading

- Topics
  - Operator Overloading
    - +, =, ==, [], (),<<
  - Destructors
  - Copy Constructor

Operator Overloading

- string s1 = ”Hello”;
- string s2 = “World”;
- string greeting = s1 + “ “ + s2;
- What is the +
    - Concatenation
- 3 + 4 == 34 ???
    - Hah no, here it means addition
- The + is a **function** that takes two parameters and does something with them. With strings, it concatenates, with numbers it adds, with other datatypes… it does whatever we want!
    - More precisely, + is an *overloaded* class method.
    - We can define the + operator (ie, the *overloaded class method*) to do whatever we think makes sense with our datatype.

Why Operator Overloading

- Multiple functions with the same name, but different parameters.
- Allows us to write code that is “cleaner”.

Overloading With Vectors

```
MyVector<int> f1 {1,2,3};
MyVector<int> f2 {1,2,3};
MyVector<int> f3 = addVectors(f1,f2);
```

```
f3 = f1 + f2; // If we overload the + operator,
we can use this syntax which is much cleaner.
```

- What is the difference between these two:
  - operator+(f1,f2) <- function
  - f1.operator+(f2) <- method
- You can create the + operator either way, but we will focus primarily on methods.

How to Create a Plus (+) *Function*

- Since + is actually an overloaded class method, and a method is just a function, we can write a function to perform the appropriate operations.
- Before looking at methods, let’s look at the standalone function version.
- The syntax for these function signatures takes a second or two to understand:
- `MyVector operator+( const MyVector & v1, const MyVector & v2 );`
- <return type>   function_name   (parameters) // but it follows the same pattern we always use.
  - Returns a MyVector
  - Is named “operator+” (though we will use it differently than a normal function)
  - Takes in two vectors
  - Put declaration in header, definition in cpp, but not belonging to a class.

Create the + *Method*

- Now, looking at the + operator as implemented as a method of our MyVector class:

- MyVector MyVector::operator+(const MyVector& rhs)

```
{
    MyVector newVec;
    return newVec;
}
```

Operators +=, -=, *=, /= Etc.

- type1& operator +=(const type2& rhs);
- Note, type1 and type2 do not have to be the same.
- For example, you might want to add an integer to a vector:
  - MyVector f1 {1,2,3}; f1 += 10;
- The return type seems to be a bit strange. You can technically write code like:
  - f1 = (f2 += f3); // This is equivalent to the following two statements:
  - f2 += f3;
  - f1 = f2;
- To handle the return type, the implementation of each of these operators ends with:
  - return *this; // A reference to the LHS object.

Overloading [ ]

- The square brackets [ ] are technically also an operator. And as such they can be overloaded.
  - This is how vector and string are made to work like arrays (even though they are objects).
- Note: you can use the [ ] on both the right and left side of an equation…
  - myVec[0] = 7;
  - int i = myVec[0];
  - const int i = myVec[0]; // What does this do and what operator signature does it required?
- Type & operator[ ](int index);
- const Type & operator[ ](int index) const; // Why this one?

Operator()

- Perhaps surprisingly, parentheses can also be used as an operator.
- This allows you make an object act like a function.
- Let’s say we have a matrix (a table).
- Matrix mat (10,10); //Does this require an operator overload?
    - No! Constructor!
- What about this?
    - float value = mat (3, 4); // Give me the value at row 3, column 4.
- This would be declared like:
    - float Matrix::operator()(int row, int col);

operator <<

- Must implement operator << as a function, it cannot be a method.

```
ostream & operator<<( ostream & out, const MyVector
& f )
{
     for (const int& d: f)
          out << d;
     return out;
}
```

- If a standalone function needs to access member vars, you typically add on the “friend” keyword.
- Why are we returning *out*? Lets us combine!
  - cout << f1 << f2 << f3;
  - ((cout << f1) << f2) << f3)

Tricky (Pointer) Business

```
std::vector<int> v1( 8 );

{
   std::vector<int> v2 = v1; // Copies the fields in v1 into v2
} // end block
```

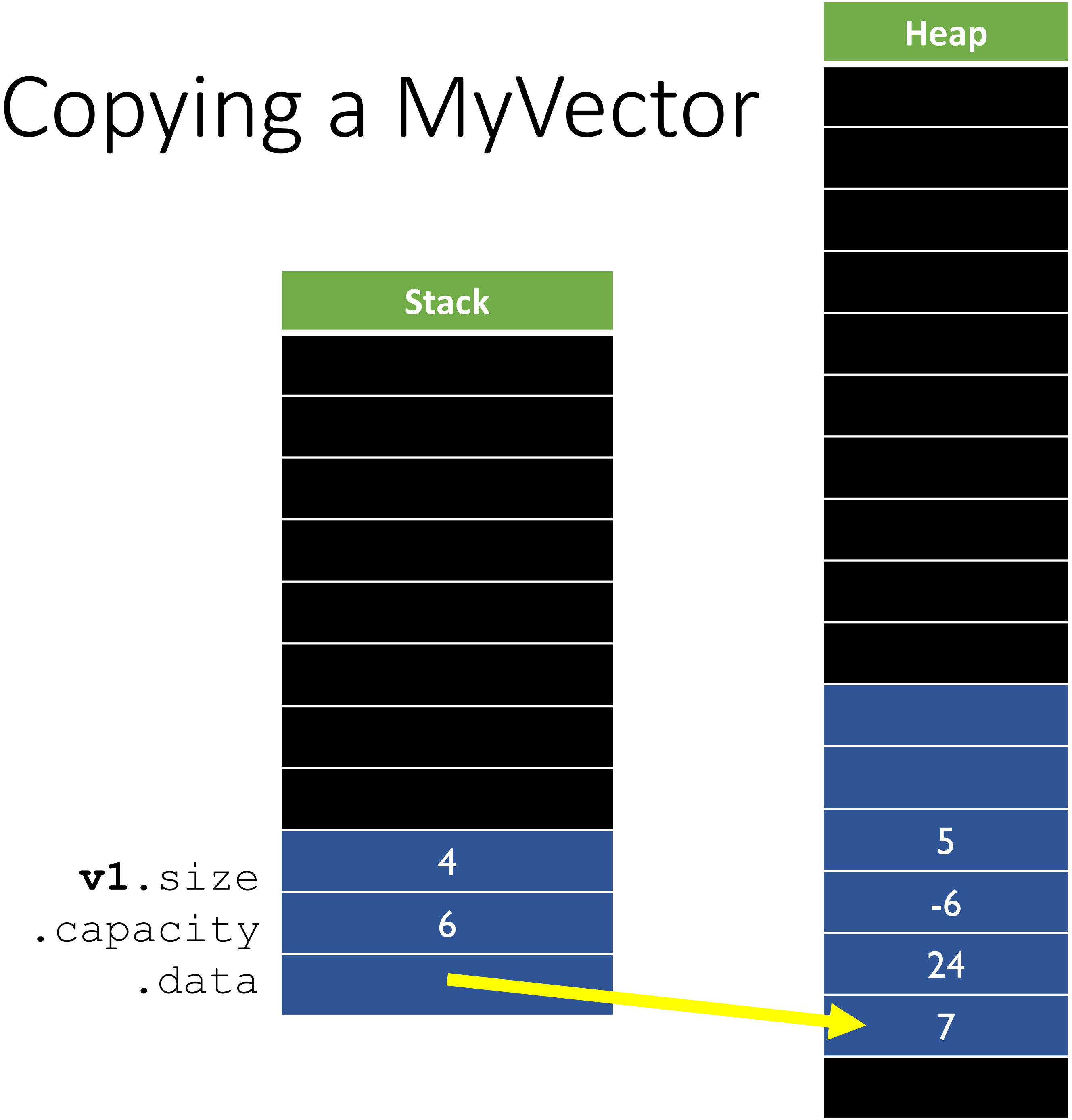
- What fields were copied?
  - size, capacity, data *
- Was the actual data copied?
  - No, just the pointer to it.
- What happens after the } (end of block) above?
- Is v2 still visible after the block ends?
  - No
  - In fact, it “goes out of scope”. What happens when a variable goes out of scope?
    - Its destructor is called.

Tricky (Pointer) Business

```
Vector v1( 8 );
{
   Vector v2 = v1; // Copies the fields in v1 into v2
} // end block
```

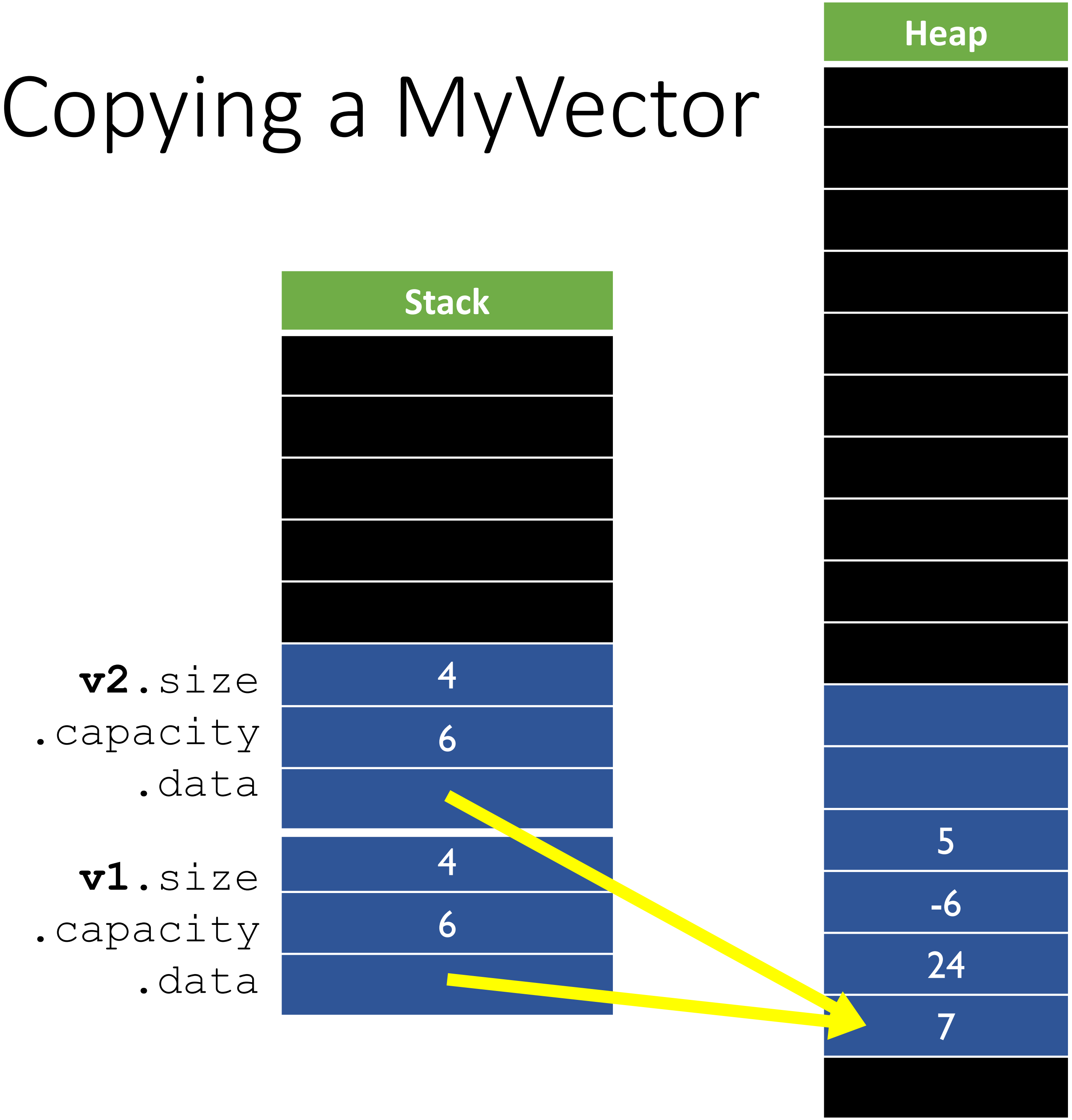
- What does the Vector's destructor do with the memory it points to?
  - Deletes it to return memory to the system and avoid a memory leak.
- What is the "tricky" problem? Consider both variables v1 and v2.
  - When v2 was destroyed, it deleted the memory that v1 is still pointing at!
    - The actual data was shared because only the pointer to the data was copied.
- See following slides for a pictorial view of this issue.

Copying a MyVector



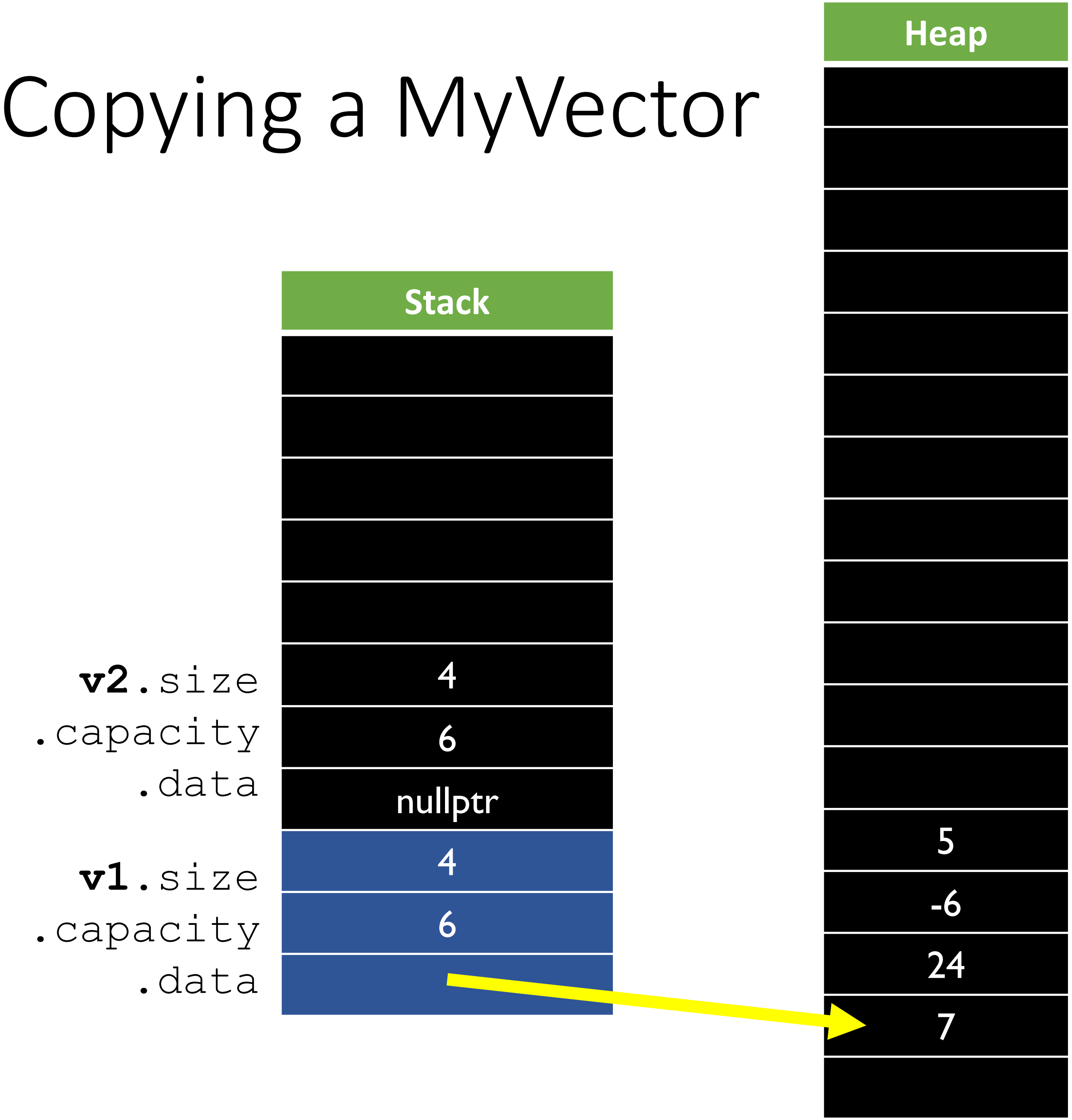
- `MyVector v2 = v1;`
    - v2 becomes a copy of v1

Copying a MyVector



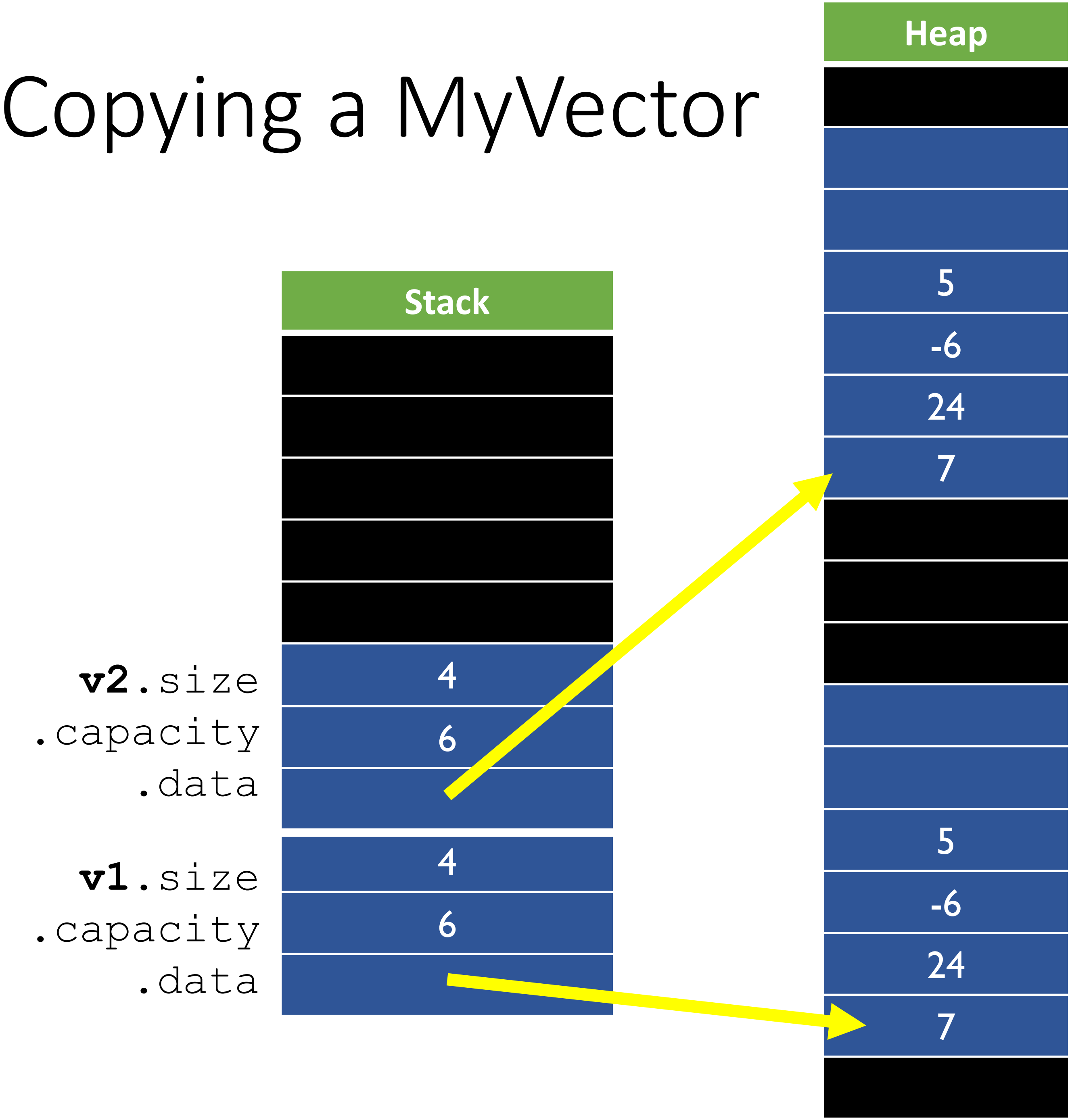
- But (in this example) we weren't smart and we didn't actually copy the vector's data.
    - We made a "Shallow Copy"
- Now what happens when v2 goes out of scope?
- The destructor is called which does something like:
    - delete(v2.data)
    - v2.data = nullptr;

Copying a MyVector



- But what about the memory that v1 thought it had?
  - It's gone (as far as the system is concerned).
  - Any access of its data that v1 does can cause strange memory issues.
- What is the solution to this problem?
  - When we copy v1, we need to *make a copy of* its data. Not just a copy of the pointer to its data.

Copying a MyVector



- v2 should allocate its own memory and copy value into it.
- This is what memory should look like if v1 and v2 are working properly.
- Note, once v1 is copied into v2, they are separate variables and any change to either one will (and should) not have an effect on the other.
- For this sort of correct copy, we need a **copy constructor and/or an operator=.**

The Copy Constructor

- The *copy constructor* takes an object of the same type as its parameter (by const reference).

  MyVector(const MyVector & original); // Example copy constructor declaration

- This constructor is called when we create a new object, eg:
  - MyVector v2(v1);
  - MyVector v3{ v1 };
  - MyVector v4 = v1;
- Within the code we write for the MyVector copy constructor, we can make a “deep copy” of the data from the MyVector we are copying.

Operator =

```
MyVector v1, v2;
v2 = v1; // While this might look like the copy constructor seen previously,
         // because v2 is not being created on this line, operator= is used.
```

- Syntax to declare:

MyVector & operator=(const MyVector & rhs);

- Remember, like all the other operators with “=“ in their name, this function will `return * this;`

What’s the Difference Between…?

Fraction f0;

f0 = Fraction(99, 100);

- and

Fraction f0(99, 100);

- How many functions are called in the first example?
  - 3
  - Default Constructor
  - Constructor that takes num, denom
  - Operator=
- How many functions are called in the 2nd example?
  - 1 – The constructor that takes num, denom

The Rule of Three

- If you have any of:
    - destructor
    - copy constructor
    - operator=
- Then you need to implement all 3.
- This guarantees that each object is created and destroyed properly.
- While the copy constructor and operator= are very similar, we have to implement both.

Constructor Initialization List

```
MyVector::MyVector( int cap ) {
        capacity_ = cap;
        size_ = 0;
        data_ = new int[ cap ];
}
```

- Using the initialization list syntax:

```
MyVector::MyVector( int cap ) : capacity_( cap ), size_( 0 ), data_( new int[ cap ] )
{
}
```

- For complex classes (classes that contain other objects), using the initialization list guarantees that every member variable has a constructor called before you get to the opening { of the constructor.
- For objects that don’t have a 0-parameter constructor (default constructor), you must call a constructor in the initialization list.
- For objects with expensive constructors, you SHOULD call the constructor in the initialization list for efficiency reasons.

Assignments

- Code Review
- Homework – Add operator overloading to you vector class!
  - If you’re done with yesterday’s HW.
  - Try at least one operator today ☺