Constructors and Assignment

Cristian Cibils
(ccibils@stanford.edu)

Evil Hangman Feedback

- Evil Hangman is still being graded but I wanted to cover some key issues people had

```
int getInteger(string prompt);
double getPositiveReal(string prompt);
char getLetter(string prompt);
string getFilename(string prompt);
```

Evil Hangman Feedback

- Instead of having a separate method to get every type, just have a templated getType

```
template <typename T>
T getType(string prompt);
```

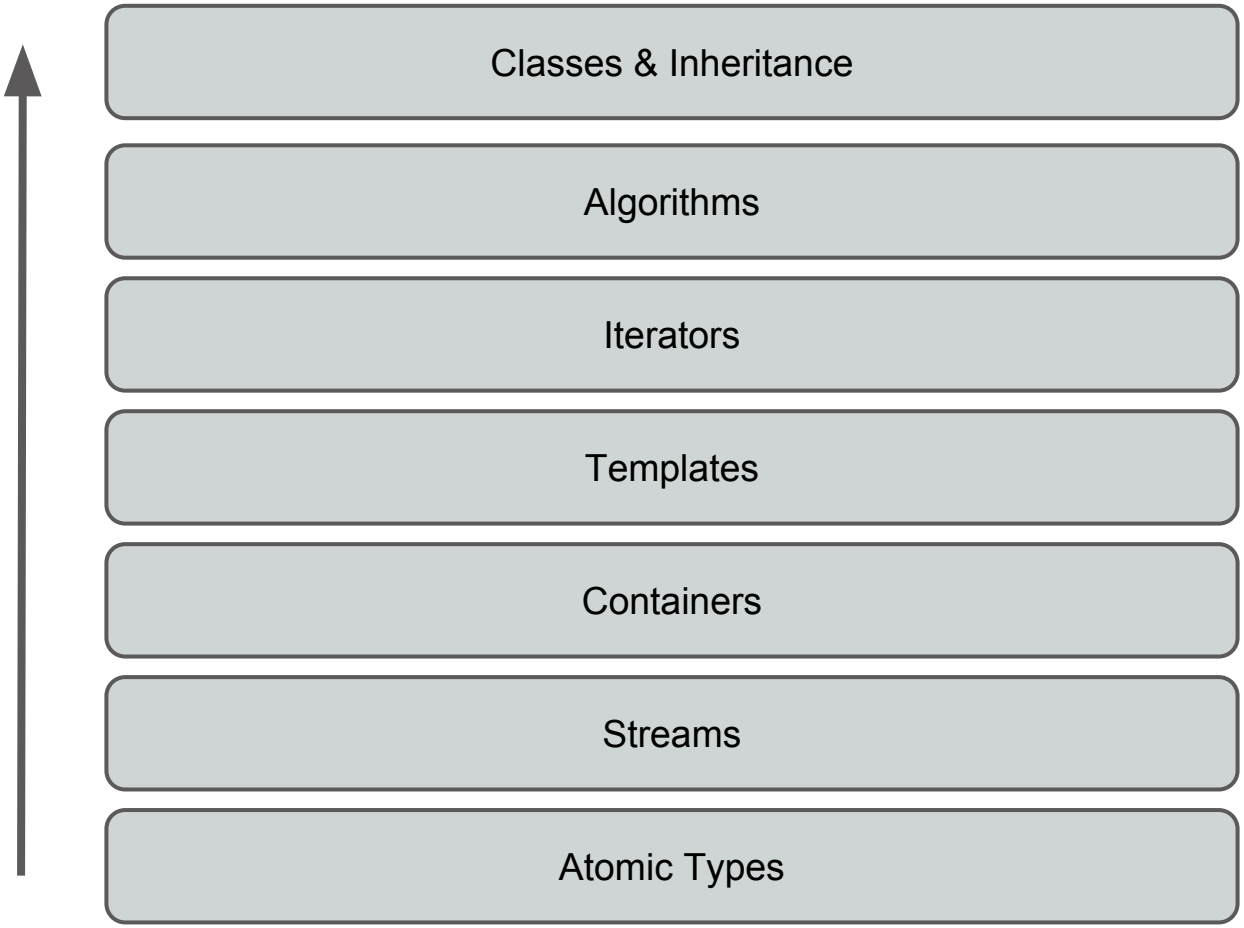
Evil Hangman Feedback

- File reading woes. Instead of having an infinite loop with a break, loop over the file itself or use copy

```
string line;
while (getline(infile, line)) {

copy(istream_iterator<string>(infile),
     istream_iterator<string>(), dest);
```

The Design of C++



Classes & Inheritance

- A bit of a monster
  - Designing Types
  - Designing Templated Types
  - Const Correctness
  - Operator Overloading
  - More fun stuff

Why Constructors?

- A constructor is a function which is called when an object is first created
  - Objects are created on the stack by a variable declaration
  - Objects on the heap are created with `new`
- The constructor sets up the initial state of the object for later functions
- This should be familiar, but let's go a bit more in depth...

Why Constructors?

```
class Vector {
   Vector() {
      logicalSize = 0;
      allocatedSize = 8;
      elems = new int[allocatedSize];
   }
};

// Both of these lines call the constructor
Vector x;
Vector *y = new Vector();
```

Why Constructors?

Why do objects have constructors?

Can't we just use an `init` function which does the same thing the constructor does?

Why Constructors?

```
struct foo {
  int value;
  void init(int v) {value = v;}
};

foo x;
x.init(42);
cout << x.value << endl;
```

Why Constructors?

- Issue #1: What if we forgot `x.init()`?

```
struct foo {
  int value;
  void init(int v) {value = v;}
};

foo x;
// x.init(42);
cout << x.value << endl; // what comes out?
```

Why Constructors?

- Issue #2: I'm incredibly lazy.

```
int main() {
   // Construct *and* initialize in one line
   HasAConstructor x(42);
   // .init() requires two lines!
   HasInitFunction y;
   y.init(42);
}
```

Why Constructors?

- Issue #3: Const data members

```
struct ConstMember {
   const int value;
   void init(int v) {value = v;}
};

ConstMember x;
x.init(42); // Error: assignment to const!
```

Why Constructors?

The notion of **initialization** is fundamental to the C++ language and distinct from the notion of **assignment**.

Why Constructors?

Initialization transforms an object's initial junk data into valid data.

Assignment replaces existing valid data with other valid data.

Why Constructors?

Initialization is defined by the **constructor** for a type.

Assignment is defined by the **assignment operator** for a type.

Constructors

We will be looking at three kinds of Constructors today

- Default Constructors
  - What you are used to but with some new tricks
- Copy Constructors
  - Construct an instance of a type to be a copy of another instance
- Copy Assignment
  - Not really a constructor
  - Assign an instance of a type to be a copy of another instance

Why Constructors?

```
// Initialization: Default Constructor
Widget x;
// Initialization: Copy Constructor
Widget y(x);
// Initialization: Copy Constructor (form 2)
Widget z = x;
// Assignment: Copy assignment
z = x;
```

Quick Note

It is not always necessary to define all types of constructors and assignment. If you don’t the compiler will create a default version for you

Constructors

We will be looking at three kinds of Constructors today

- Default Constructors
  - What you are used to but with some new tricks
- Copy Constructors
  - Construct an instance of a type to be a copy of another instance
- Copy Assignment
  - Not really a constructor
  - Assign an instance of a type to be a copy of another instance

Default Constructors

- A constructor looks just like any other member function for a type, with 3 distinctions
  - Constructors have **no return value** listed (not even void)
  - Constructors have the **same name as the type** in question
  - Constructors can have an **initialization list**

Default Constructors

Initialization lists allow us to **initialize** (not assign) data members when we initialize our type.

```
// Assignment
struct Widget {
    const int value;
    Widget();
};
Widget::Widget() {
    value = 42; //ERROR
}
```

```
// Initialization
struct Widget {
    const int value;
    Widget();
};
Widget::Widget()
    : value(42) {}
```

Default Constructors

Initialization lists can have multiple parts

```
struct Person {
   int age;
   string name;
   Person();
};
Person::Person():age(36), name(“Kanye”) {
   //Empty constructor since nothing to assign
}
```

Default Constructors

Constructors solve all 3 of the problems with the init function. Value is initialized to v, not assigned

```
struct ConstMember {
   const int value;
   ConstMember(int v) : value(v) {}
};
int main() {
   ConstMember b(42); // value is 42
}
```

Vector Constructors

Let's now take a look at a more complex constructor -- our old friend Vector<T>.

Vector Constructors

```
// Initialize an empty vector:
vector<int> a;

// 42 elements: all zero
vector<int> b(42);

// 42 elements: all set to 11
vector<int> c(42, 11);
```

Interlude: Default Parameters

We’re about to do something cool, but we need to review default parameters first.

Interlude: Default Parameters

- In C++, we can list **default parameters** for functions which take arguments
- Functions without default parameters can have their rightmost parameters omitted and the default values will be used
- The syntax is simple, but default parameters should only be listed in the *declaration* of a function, not the *definition*

Interlude: Default Parameters

```
// Declare our default arguments
void f(int a, int b = 5, int c = 42);

// Define our function
void f(int a, int b, int c) {
   cout << a << " ";
   cout << b << " ";
   cout << c << endl;
}
```

Interlude: Default Parameters

```
void f(int a, int b = 5, int c = 42);
void f(int a, int b, int c) {
   cout << a << " ";
   cout << b << " ";
   cout << c << endl;
}

f(1);       // 1 5 42
f(1,2);     // 1 2 42
f(1,2,3);   // 1 2 3
```

Vector Constructors

Let's try implementing the default and fill constructors in one step using default parameters.

Constructors

We will be looking at three kinds of Constructors today

- Default Constructors
  - What you are used to but with some new tricks
- Copy Constructors
  - Construct an instance of a type to be a copy of another instance
- Copy Assignment
  - Not really a constructor
  - Assign an instance of a type to be a copy of another instance

Copy Constructors

- A copy constructor is called when an instance of a type is constructed from another instance
- Two ways it can be called

```
//directly call the copy constructor
int x(4);

//implicitly call the copy constructor
int y = 2;
```

Copy Constructors

- A copy constructor is called when an instance of a type is constructed from another instance

```
Vector<string> a(10, “hi”);
//directly call the copy constructor
Vector<string> b(a);

//implicitly call the copy constructor
Vector<string> c = a;
```

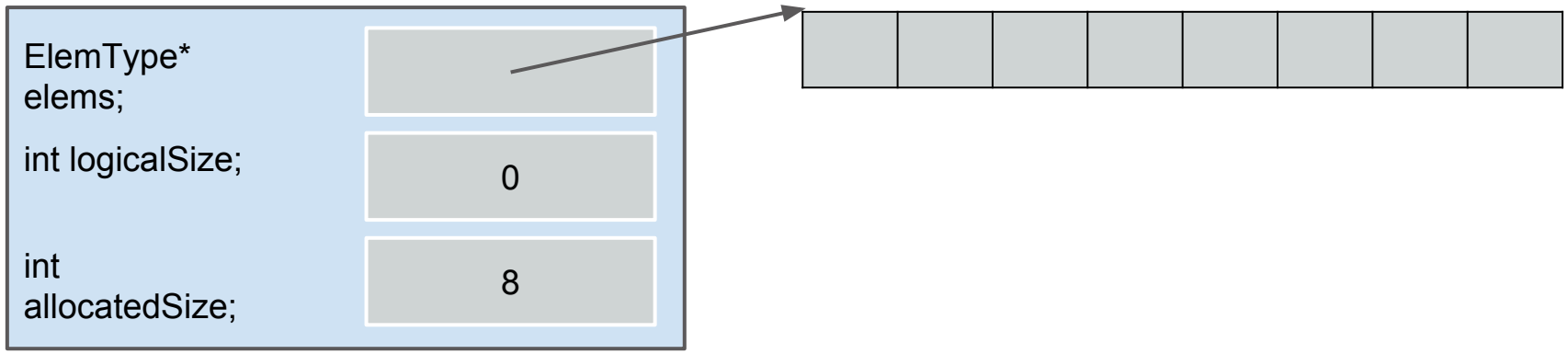
Copy Constructors

Before writing the copy constructor, let's think about how we're going to write it.

- First idea: just copy all of their member variables
- We'll have the correct size and element pointer, so this works right?
- This is what the default copy constructor does

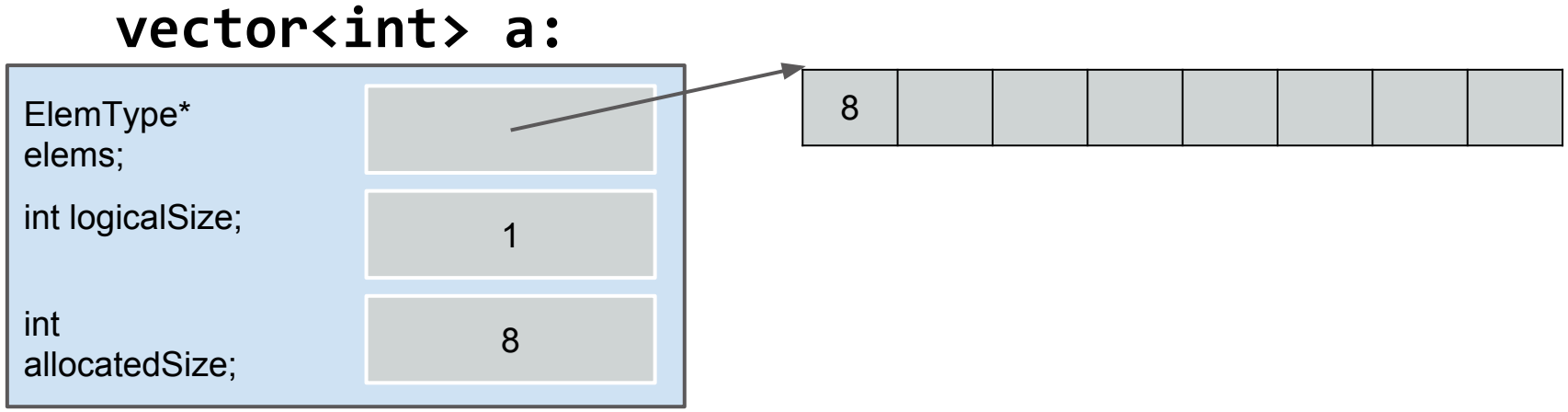
Copy Constructors

vector<int> a:



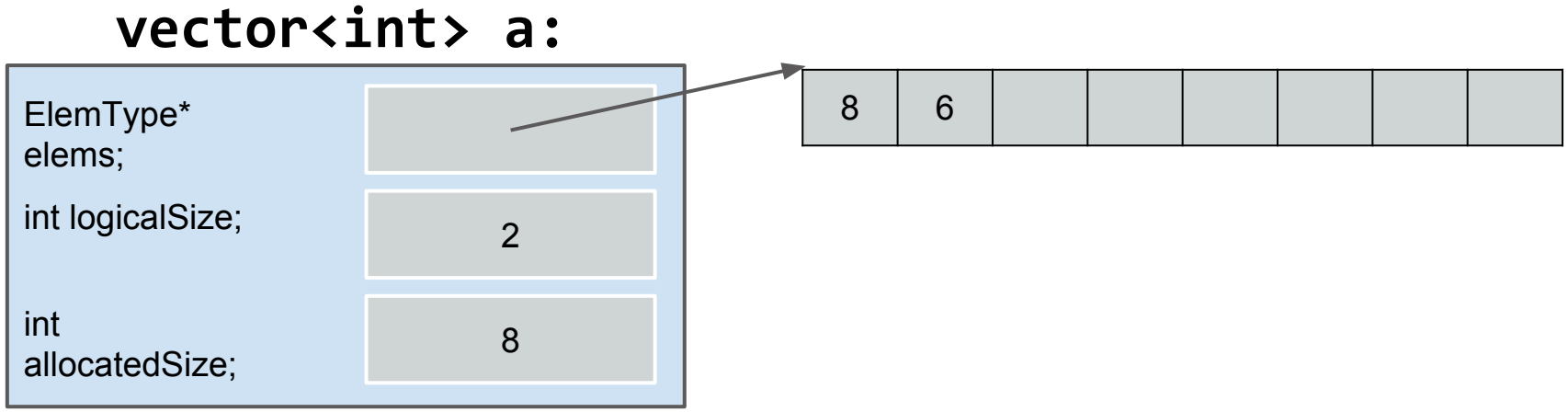

```
Vector<int> a;
```

Copy Constructors



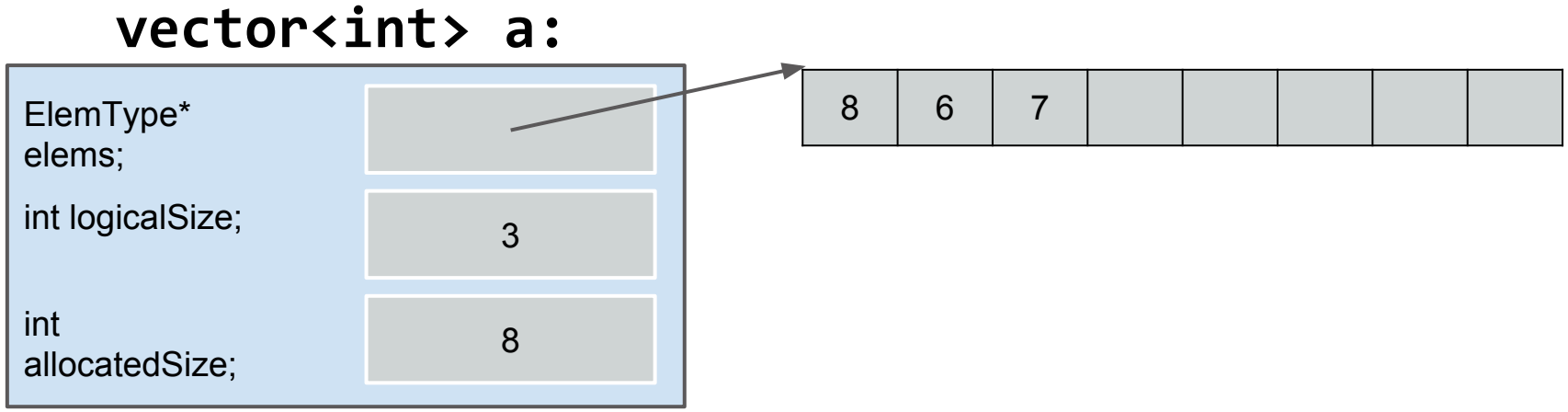

```
Vector<int> a;
a.push_back(8);
```

Copy Constructors



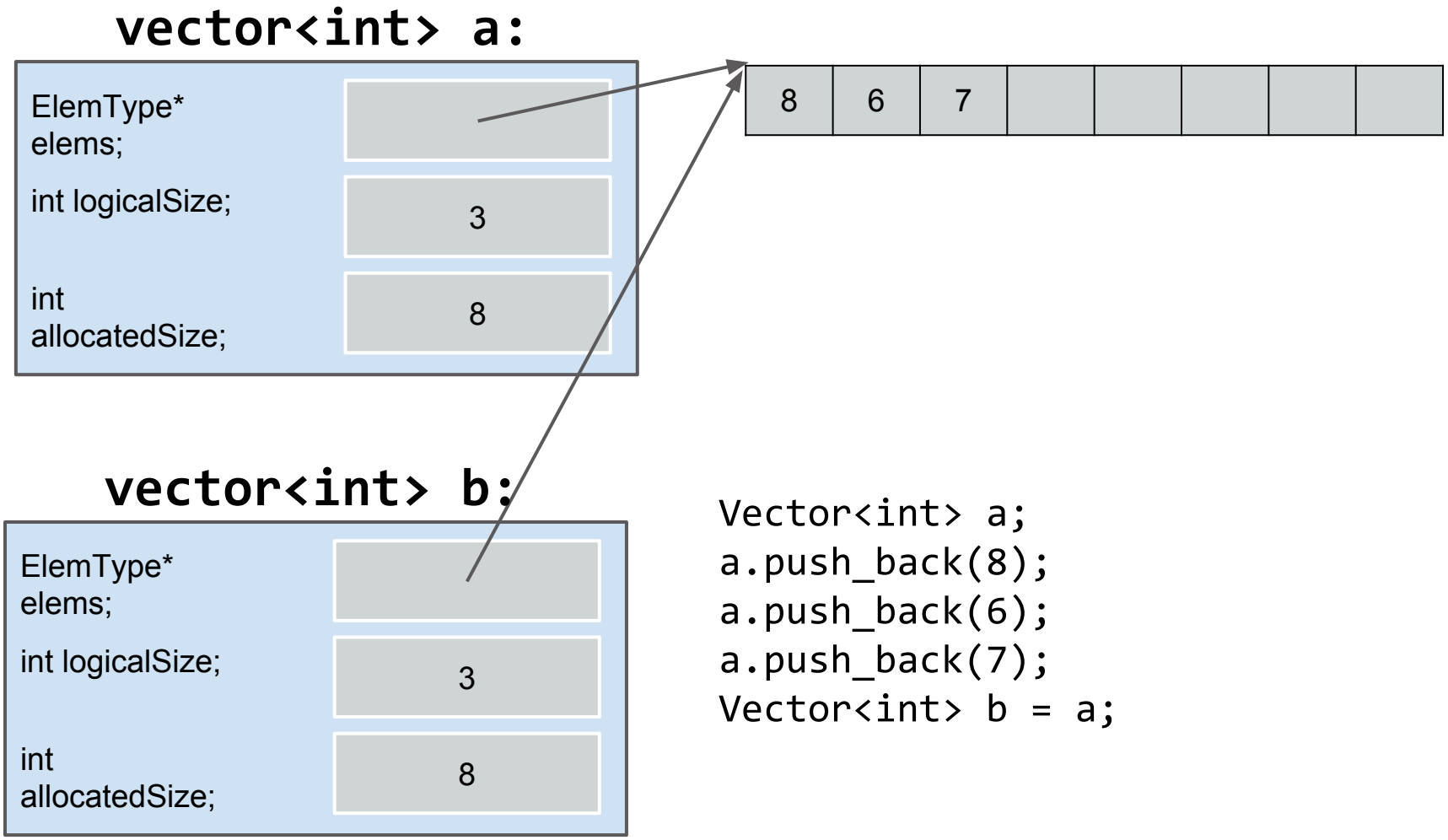

```
Vector<int> a;
a.push_back(8);
a.push_back(6);
```

Copy Constructors



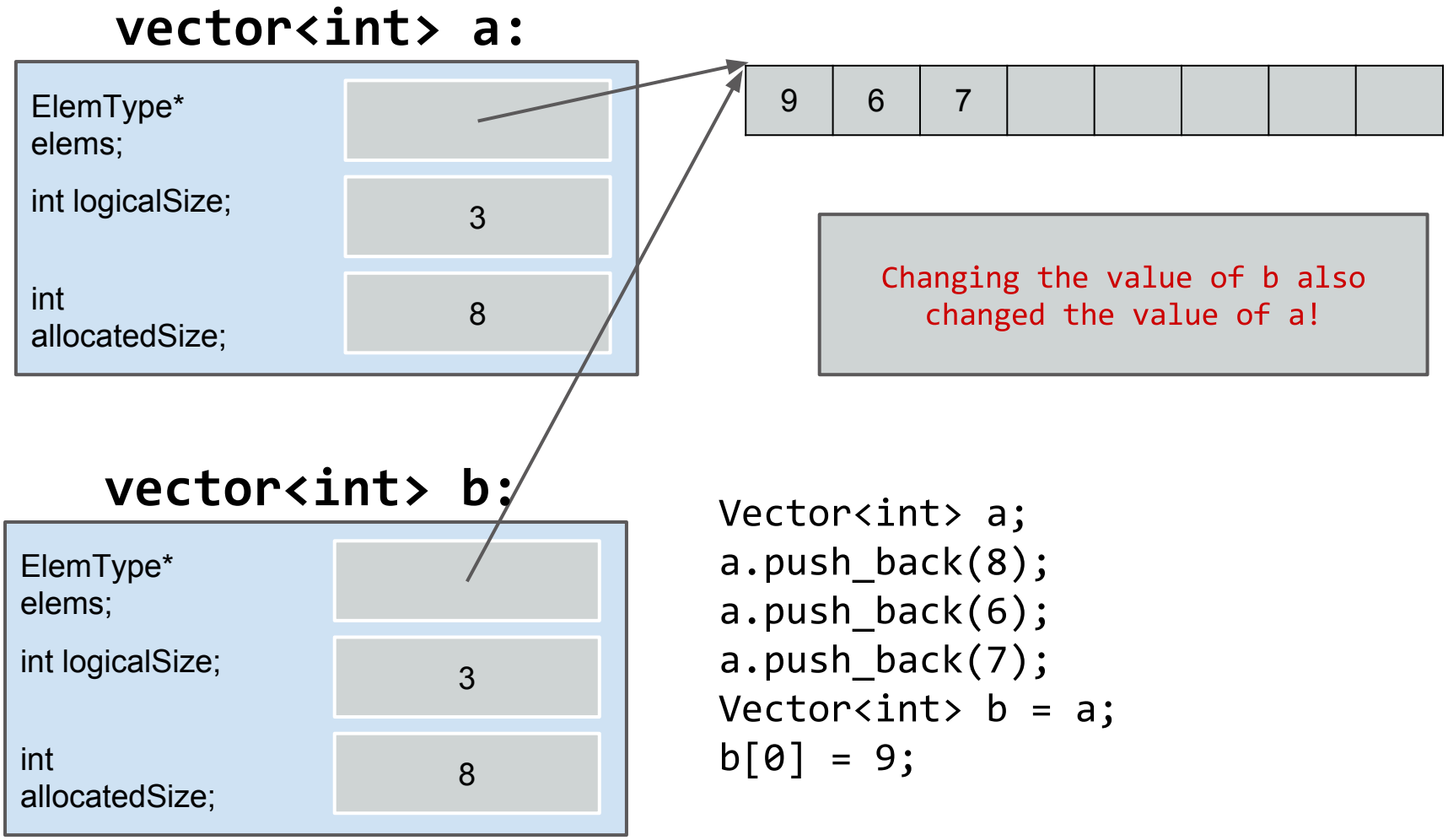

```
Vector<int> a;
a.push_back(8);
a.push_back(6);
a.push_back(7);
```

Copy Constructors



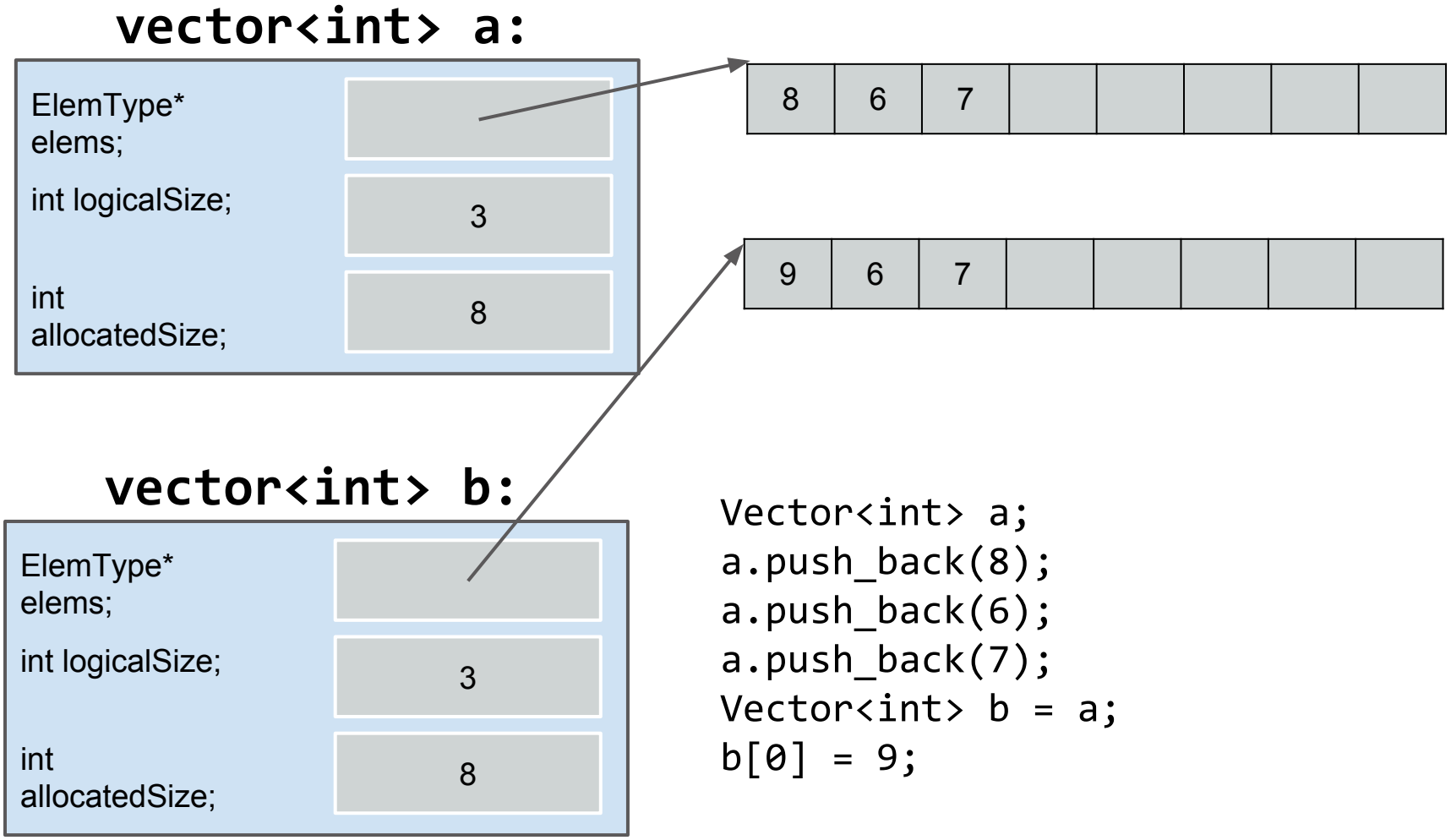

```
Vector<int> a;
a.push_back(8);
a.push_back(6);
a.push_back(7);
Vector<int> b = a;
```

Copy Constructors




```
Vector<int> a;
a.push_back(8);
a.push_back(6);
a.push_back(7);
Vector<int> b = a;
b[0] = 9;
```

Copy Constructors




```
Vector<int> a;
a.push_back(8);
a.push_back(6);
a.push_back(7);
Vector<int> b = a;
b[0] = 9;
```

Copy Constructors

If you have pointer member variables, you should always define a copy constructor

Copy Constructors

Let's try implementing a proper copy constructor for vector in which we copy over the elements

Constructors

We will be looking at three kinds of Constructors today

- Default Constructors
  - What you are used to but with some new tricks
- Copy Constructors
  - Construct an instance of a type to be a copy of another instance
- Copy Assignment
  - Not really a constructor
  - Assign an instance of a type to be a copy of another instance

Copy Assignment

- Now that we know how to write constructors, include the copy constructor, we're ready to move on to the copy assignment operator
- Remember, assignment takes an already initialized object and gives it new values

```
int x = 4; //Copy Constructor
x = 2; //Copy Assignment
```

Copy Assignment

The syntax for copy assignment is as follows. Note that this is the same syntax as any other operator overload.

```
class Widget {
public:
    Widget& operator=(const Widget& other);
//Other member vars and functions
};

Widget& Widget::operator=(const Widget& other) {
    // Code to copy data from other
}
```

Copy Assignment

Implementing the **copy assignment operator** is tricky for a couple of reasons:

- Catching memory leaks
- Handling self assignment
- Understanding the return value

Copy Assignment

I don't want to go through the gory details of how hard it is to write the truly optimal copy assignment operator.

Instead, let's use the "copy and swap" idiom to do save ourselves the trouble!

Not quite as efficient, but much, much cleaner

Copy Assignment

The **copy and swap** idiom works as follows:

- We have an existing value we want to modify, and an existing value to read data from
- Use the copy constructor to create a temporary value from the value we're reading data from
- Swap the contents of the value to modify and the temporary

Copy Assignment

```
class Widget {
    int value;
public:
    void swap(Widget& other);
    Widget& operator=(const Widget& other);
};
void Widget::swap(Widget& other) {
    std::swap(value, other.value);
}
Widget& Widget::operator=(const Widget& other) {
    Widget temp(other);
    swap(temp);
    return *this;
}
```

Copy Assignment

We can improve this function a bit by handling the copying into a temporary by using pass by value

Copy Assignment

```
class Widget {
    int value;
public:
    void swap(Widget& other);
    Widget& operator=(const Widget& other);
};
void Widget::swap(Widget& other) {
    std::swap(value, other.value);
}
Widget& Widget::operator=(const Widget& other) {
    Widget temp(other);
    swap(temp);
    return *this;
}
```

Copy Assignment

```
class Widget {
    int value;
public:
    void swap(Widget& other);
    Widget& operator=(Widget other);
};
void Widget::swap(Widget& other) {
    std::swap(value, other.value);
}
Widget& Widget::operator=(Widget other) {
    swap(other);
    return *this;
}
```

Vector Assignment

Let's take a look at how to do this in vector.

Some C++ Quirks

While we are on the topic of constructors, what does this line of code do?

```
Vector<int> v();
```

Some C++ Quirks

While we are on the topic of constructors, what does this line of code do?

```
int x(); //meant to do int x; or int x(2);
```

This is called the **most vexing parse** and has plagued many a C++ programmer. By adding the parentheses, v is treated as a function declaration.

Some C++ Quirks

In our copy constructor the following code will compile

Vector<int> x = 15;

To prevent accidental type conversions, use the **explicit** keyword before the declaration of your constructor

Some C++ Quirks

A default constructor, copy constructor, and copy assignment operator will all be defined for you if you don’t define them.

To prevent this, you can declare them in the private section of the class