14 - Constructors

Reid Watson
(rawatson@stanford.edu)

Administrivia

- Evil Hangman is due this Friday

What 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...

What 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 Widget {
  private:
  int widgetValue; // and more...
  public:
  void init(int value);
  void printValue();
};
Widget w;
w.init(42);
```

Why Constructors?

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

```
#include <iostream>
using namespace std;
struct foo {
  int value;
  void init(int v) {value = v;}
};
int main() {
  foo x;
  // x.init();
  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;}
};
int main() {
   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.

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;
```

Why Constructors?

```
// Initialization: Default Constructor
Widget x;
Widget y;
// Assignment: Copy assignment
x = y;
```

Why Constructors?

```
// Initialization: default constructor
Widget x;

// Function declaration (?!?)
// “Most vexing parse”
Widget y();
```

How 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**

How Constructors?

Initializer 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) {}
```

How Constructors?

Initialization lists can have multiple parts

```
struct Person {
   int age;
   string name;
   Person();
};
Person::Person()
   : age(36)
   , name(“Kanye”)
   {}
```

How Constructors?

```
// Constructors solve all 3 of the problems
// with the init function
struct ConstMember {
   const int value;
   ConstMember(int v = 0) : value(v) {}
};
int main() {
   ConstMember a; // value is 0
   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

```
#include <vector>
using namespace std;
int main() {
    // No elements:
    vector<int> a;
    // 42 elements: all zero
    vector<int> b(42);
    // 42 elements: all set to 11
    vector<int> c(42, 11);
    // Copy constructor
    vector<int> d(c);
}
```

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 << endl;
  cout << b << endl;
  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 << endl;
   cout << b << endl;
   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.

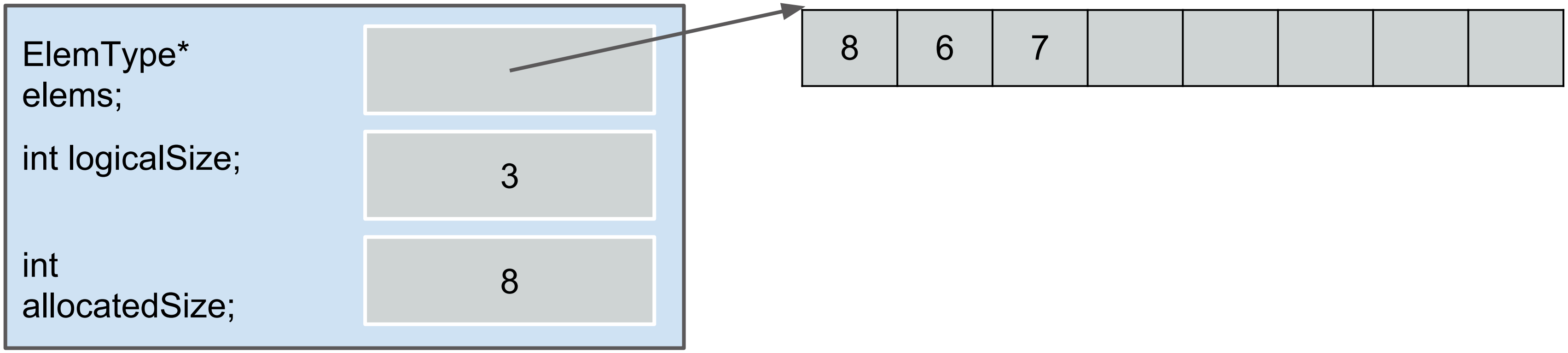
Vector Constructors

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?

Vector Constructors

vector<int> a:



ElemType* elems;

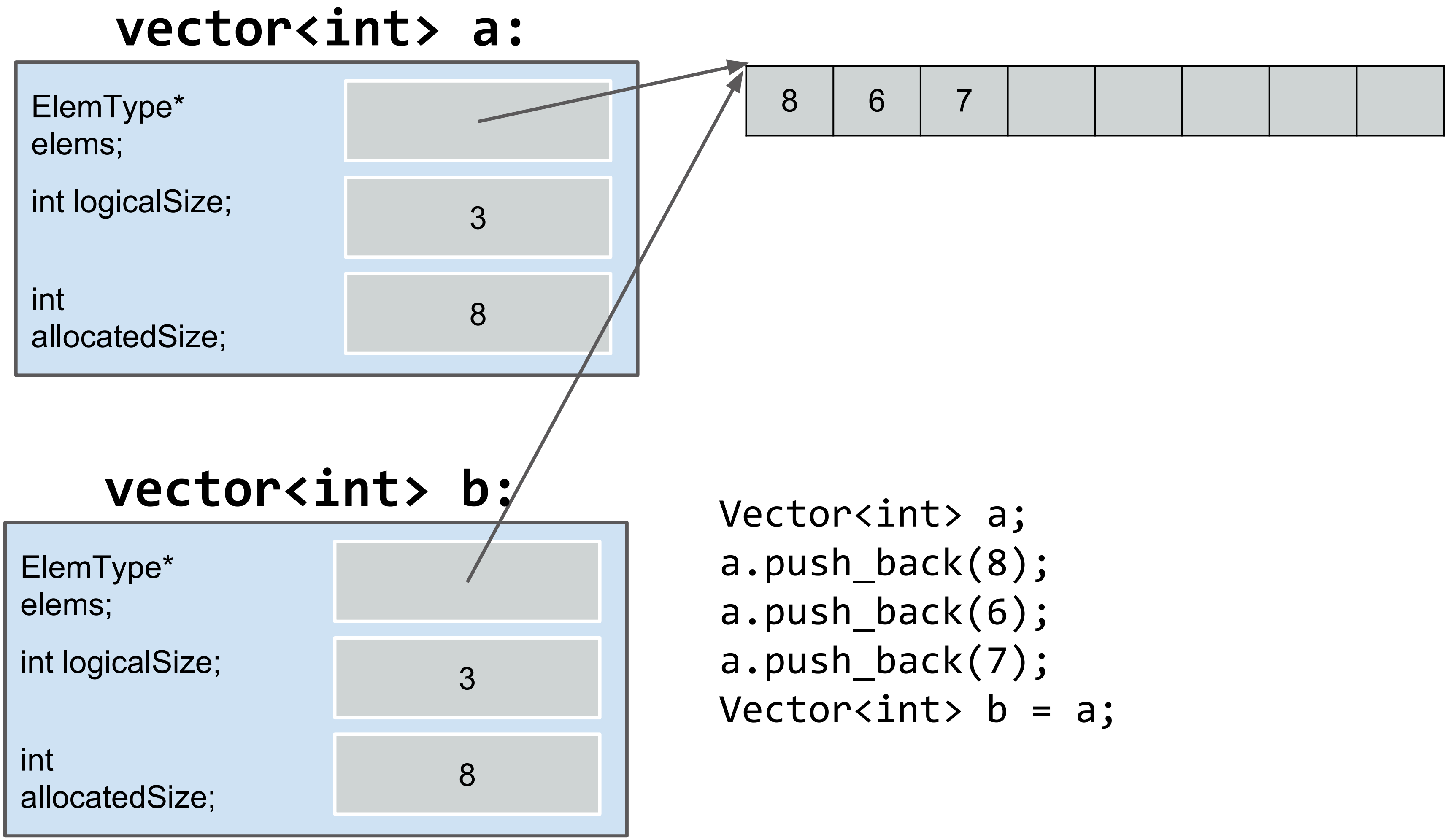
int logicalSize; 3

int allocatedSize; 8

8 6 7

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

Vector Constructors



vector<int> a:

ElemType* elems;

int logicalSize; 3

int allocatedSize; 8

8	6	7					

vector<int> b:

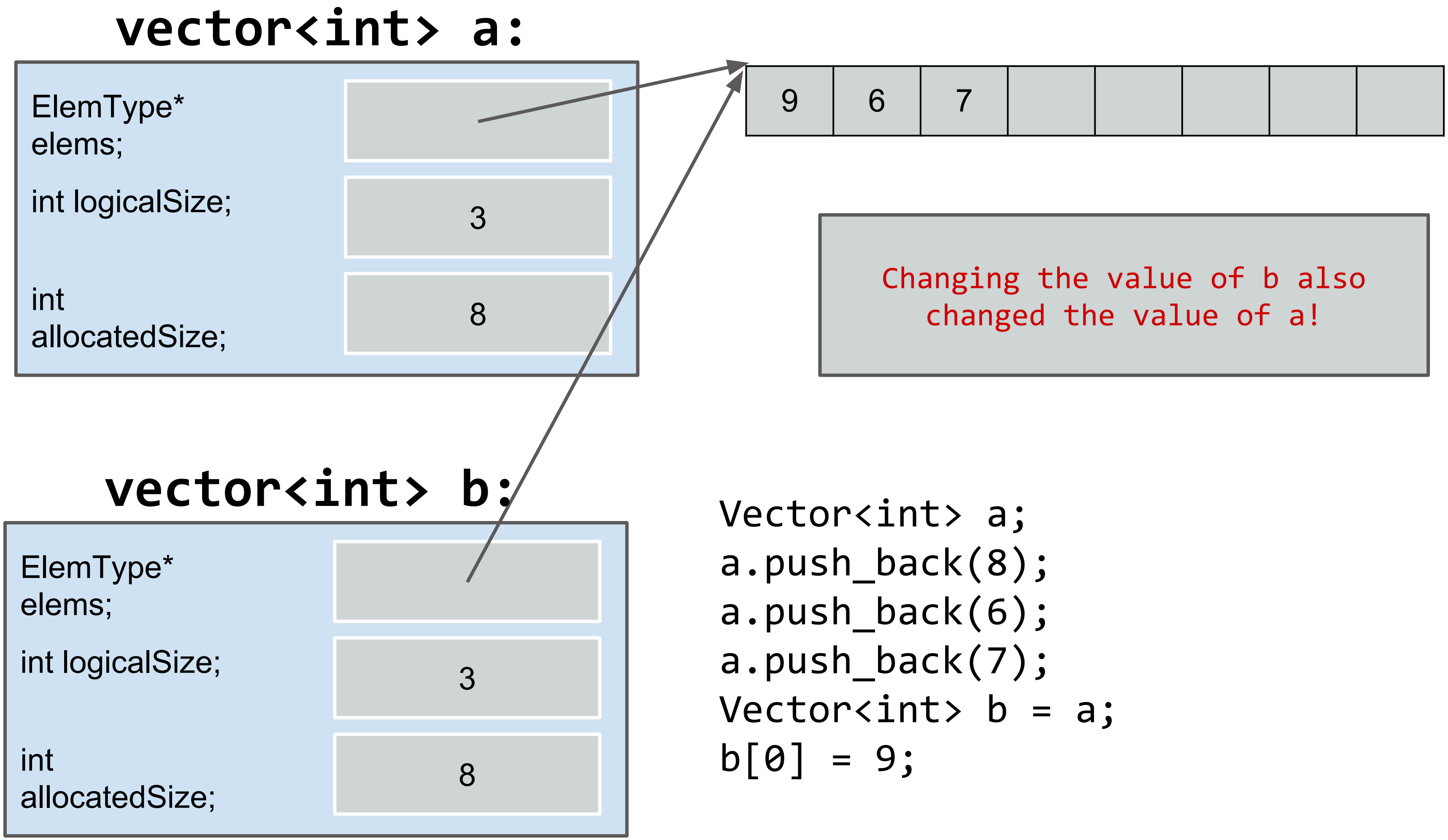
ElemType* elems;

int logicalSize; 3

int allocatedSize; 8

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

Vector Constructors



Changing the value of b also changed the value of a!

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

Vector Constructors

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

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.

Copy Assignment

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

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

Vector Assignment

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

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

Vector 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!

Vector 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

Vector Assignment

```
class Widget {
    int value;
    public:
    void swap(Widget& other);
    Widget& 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;
}
```

Vector Assignment

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

Vector Assignment

```
class Widget {
    int value;
    public:
    void swap(Widget& other);
    Widget& 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.