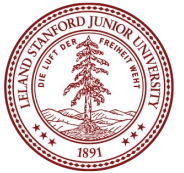
Implementing an ADT

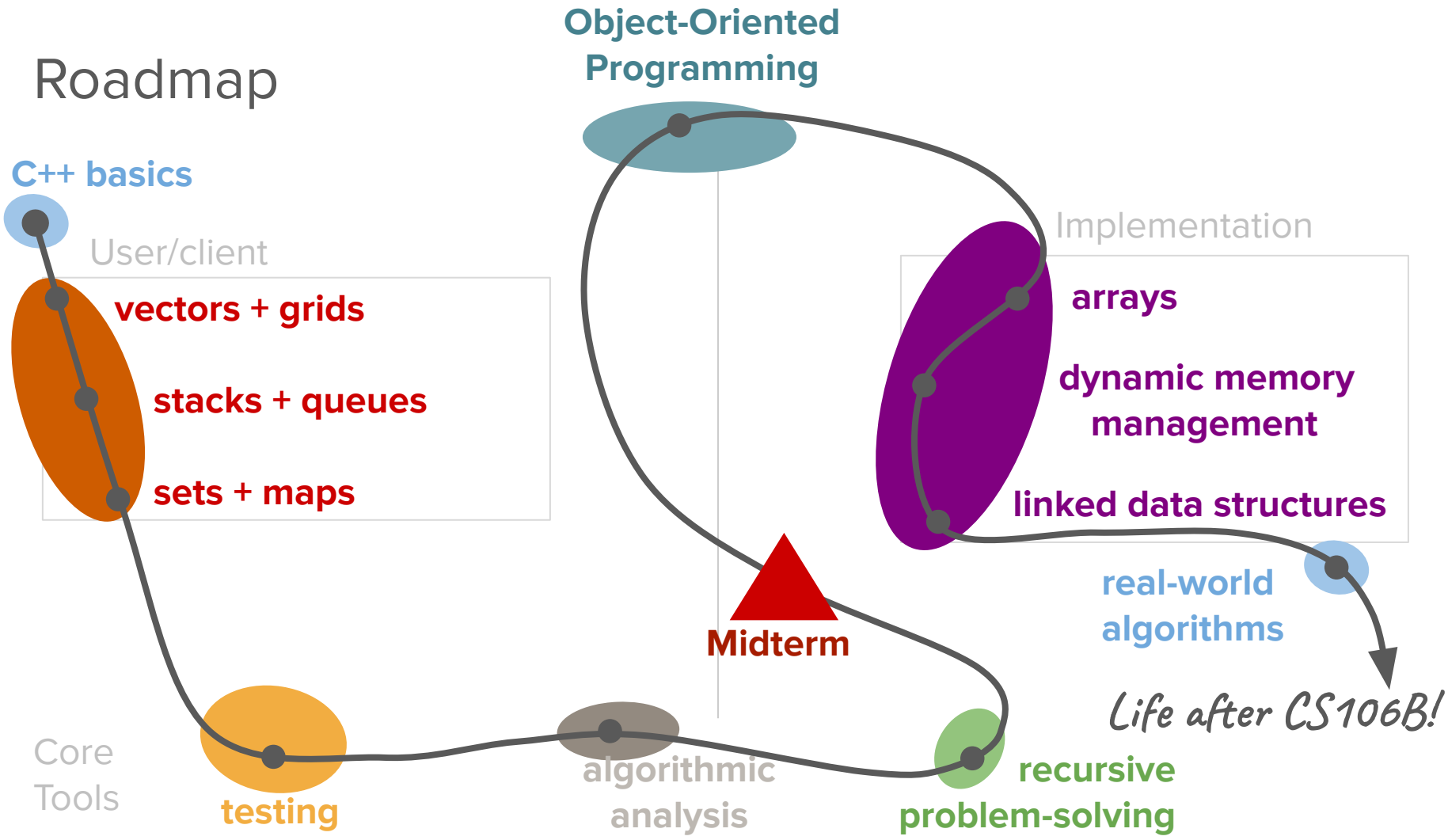
(reverse engineering a vector)

What's a CS106B topic you want to improve on after the midterm?

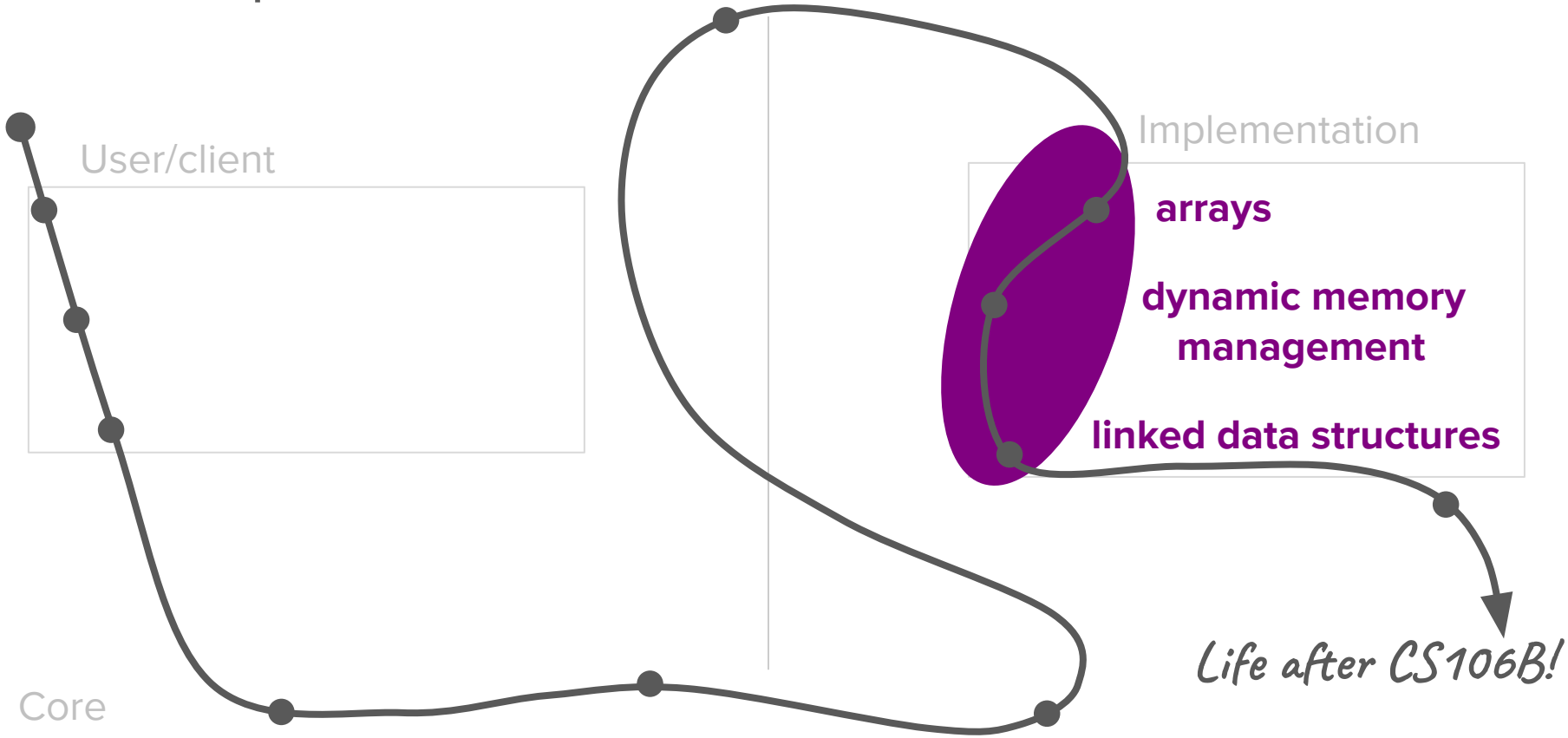
pollev.com/cs106poll



Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Midterm
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Roadmap


User/client
Implementation
arrays
dynamic memory management
linked data structures
Life after CS106B!
Core
Tools

Today's question

How can we use fundamental data storage capabilities in C++ to implement an ADT class?

Today's topics

1. Review
2. Designing OurVector
3. Visualizing OurVector Operations
4. Implementing OurVector

Review

[arrays and dynamic memory management]

Memory from the Stack vs. Heap

```
Vector<string> varOnStack;
```

- So far, all variables we've created get defined on the **stack**
- This is called static memory allocation
- Variables on the stack are stored directly to the memory and access to this memory is very fast
- We don't have to worry about memory management

```
string* arr = new string[numValues];
```

- We can now request memory from the **heap**
- This is called dynamic memory allocation
- We have more control over variables on the heap
- But this means that we also have to handle the memory we're using carefully and properly clean it up when done

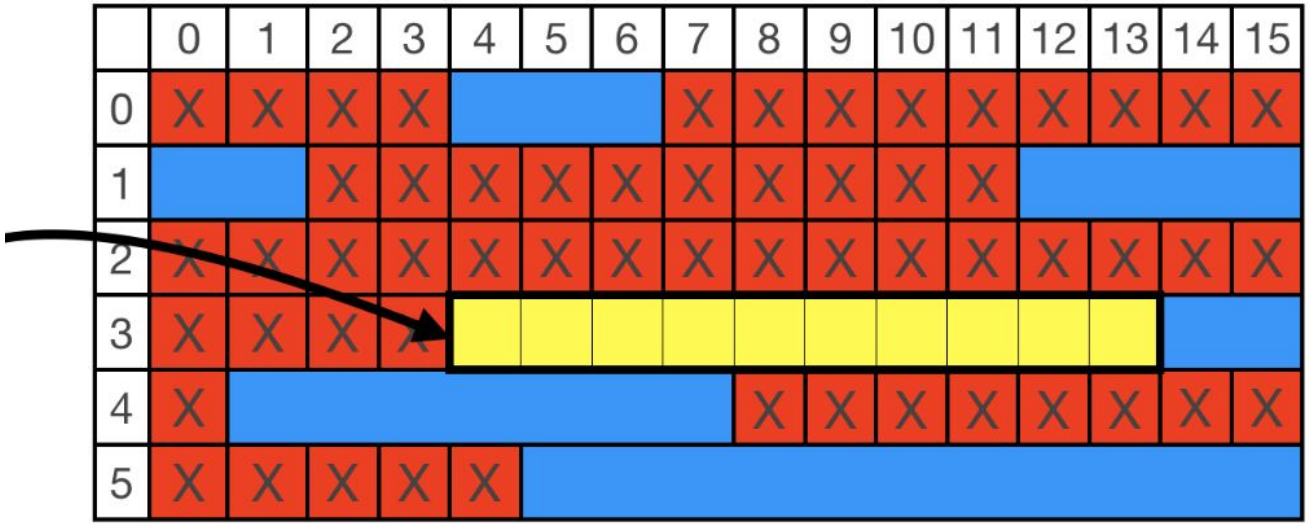
Acquiring and Using Storage Space

- Low-level storage space in C++ is acquired using **dynamic memory allocation**.

- Dynamic memory allocation normally has three steps:
  - You can, at runtime, ask for extra storage space, which C++ will give to you.
  - You can use that storage space however you'd like.
  - You have to explicitly tell the language when you're done using the memory.

```
int* tenInts = new int[10];
```

The OS will find a contiguous array for 10 integers and give you that memory back

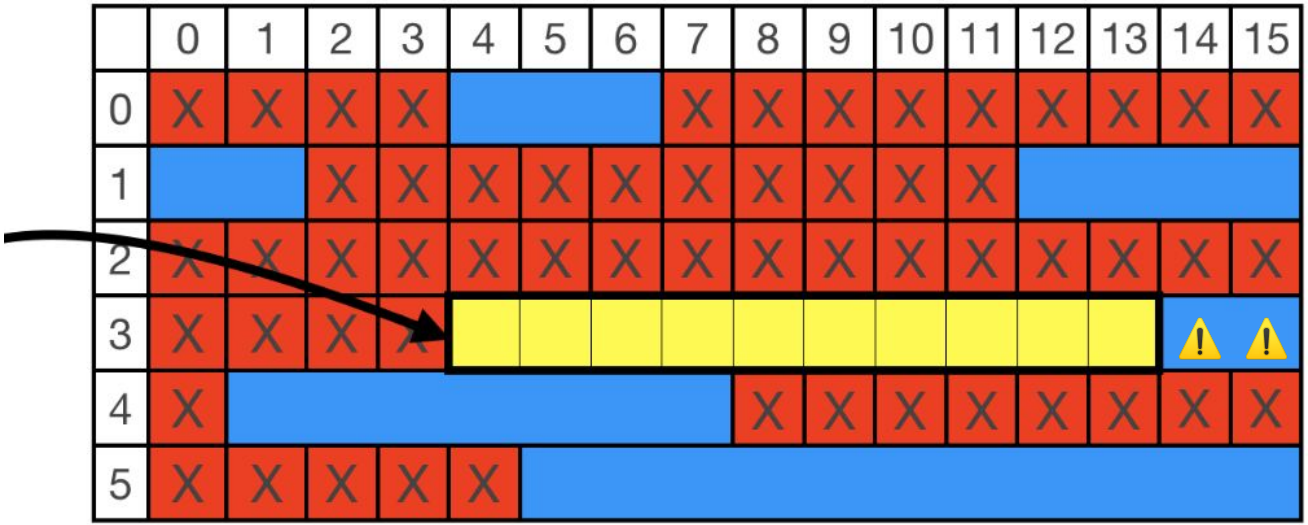


Credit: Neel Kishnani, Chris Gregg

❗❗ Don’t access off the end of your array ❗❗

```
int* tenInts = new int[10];

tenInts[10] // DON'T!!
```

Even though there’s available space at 14 and 15, it’s not yours, so you shouldn’t access it 🤷🏽‍♀️



Credit: Neel Kishnani, Chris Gregg

Arrays

- Storage space on computers, which we often refer to as memory, is allocated in organized chunks called **arrays**

- An array is a contiguous chunk of space in the computer's memory, split into slots, each of which can contain one piece of information
  - Contiguous means that each slot is located directly next to the others. There are no "gaps."
  - All arrays have a specific type. Their type dictates what information can be held in each slot.
  - Each slot has an "index" by which we can refer to it.

Index:	0	1	2	3	4	5	6

dynamic memory allocation

arrays (that we access with a pointer)

We need arrays on heap memory when designing collection classes so that...

(1) we have persistent storage over the lifetime of each class instance; available across methods

(2) We can generate storage when we need it (in the constructor)

Dynamically Allocating Arrays

- First, declare a variable that will point at the newly-allocated array. If the array elements have type `T`, the pointer will have type `T*`.
  - e.g. `int*, string*, Vector<double>*`
- Then, create a new array with the `new` keyword and assign the pointer to point to it.
- In two separate steps:

```
T* arr;
arr = new T[size];
```

- Or, in the same line:

```
T* arr = new T[size];
```

Pointers

- A pointer is a brand new data type that becomes very prominent when working with dynamically allocated memory.

- Just like all other data types, pointers take up space in memory and can store specific values.

- The meaning of these values is what's important. **A pointer always stores a memory address**, which is like the specific coordinates of where a piece of memory exists on the computer.

- Thus, they quite literally "point" to another location on your computer.

Properties of Dynamically Allocating Arrays

- The array you get from **`new[]`** is **fixed-size**: it can neither grow nor shrink once it's created.
  - The programmer's version of "conservation of mass."

- The array you get from **`new[]`** has **no bounds-checking.** Walking off the beginning or end of an array triggers *undefined behavior*.
  - Literally anything can happen: you read back garbage, you crash your program, you let a hacker take over your computer, etc...

- The array you get from the **`new[]`** keyword comes from an area of memory called the heap.

Final Takeaways

- You can create arrays of a fixed size at runtime by using **`new[]`**.
- C++ arrays don't know their lengths and have no bounds-checking. With great power comes great responsibility.
- You are responsible for freeing any memory you explicitly allocate by calling **`delete[]`**. Otherwise, your program will have memory leaks.
- Once you've deleted the memory pointed at by a pointer, you have a dangling pointer and shouldn't read or write from it.

How can we use fundamental data storage capabilities in C++ to implement an ADT class?

Arrays vs. Vectors

- Notice that we access the elements of an array just like we access them in a Vector, with square brackets.

- **BUT arrays are not objects** – they don't have any functions associated with them.

- So, you can't do this:

```
int* firstTen = new int[10];
int len = firstTen.length(); // ERROR! No functions!
firstTen.add(42); // ERROR! No functions!
firstTen[10] = 42; // ERROR! Buffer overflow!
```

Arrays vs. Vectors

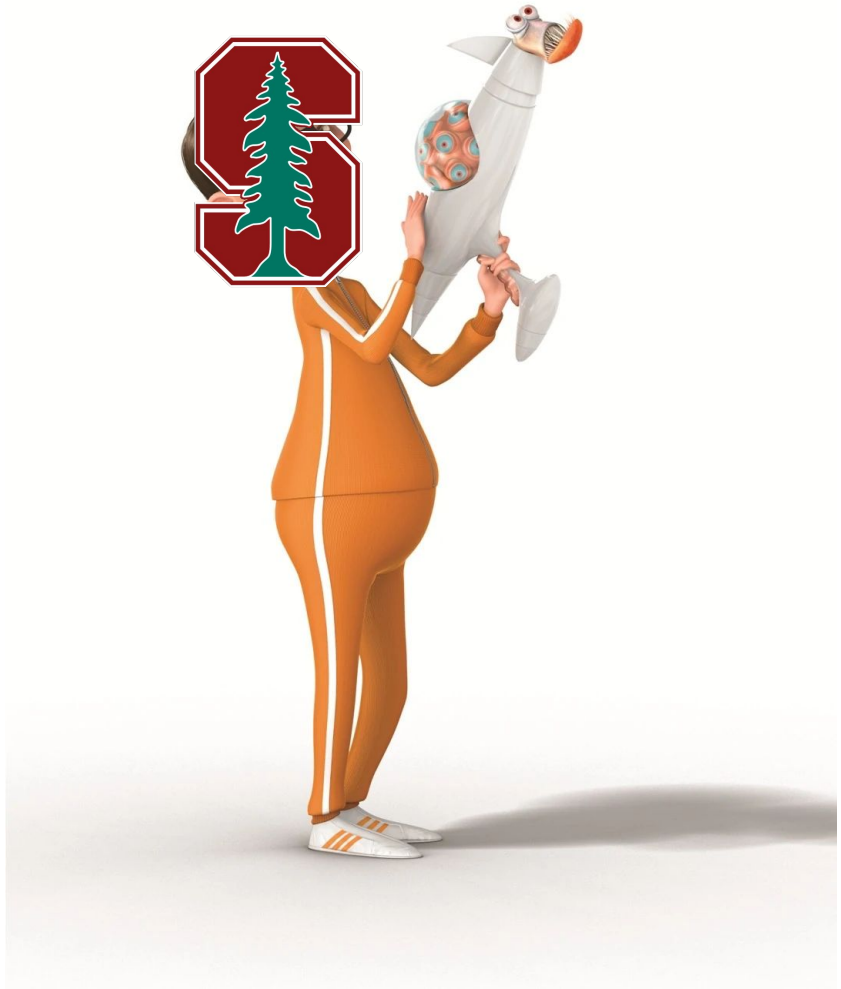
- Arrays are a very necessary tool to use if we want to actually store information in a structured way in a program.

- Vectors are a great abstraction, providing helpful methods and a clean interface that other programmers can use to solve interesting problems.

- **Idea:** Let's use a dynamically allocated array as the underlying method of data storage for a Vector class. Best of both worlds!


Always has been
Wait it's all arrays ?

Designing `OurVector`



What is `OurVector`?

- Goal: Let's make our very own version of the Stanford C++ Vector that we've been using all quarter long.
  - It all will feel so much cooler when we've built it ourselves!

What is `OurVector`?

- Goal: Let's make our very own version of the Stanford C++ Vector that we've been using all quarter long.
  - It all will feel so much cooler when we've built it ourselves!

- Scope Constraints (aka "You've Gotta Start Somewhere"):

What is `OurVector`?

- Goal: Let's make our very own version of the Stanford C++ Vector that we've been using all quarter long.
  - It all will feel so much cooler when we've built it ourselves!

- Scope Constraints (aka "You've Gotta Start Somewhere"):
  - We will only implement a subset of the functionality that the Stanford Vector provides.

What is `OurVector`?

- Goal: Let's make our very own version of the Stanford C++ Vector that we've been using all quarter long.
  - It all will feel so much cooler when we've built it ourselves!

- Scope Constraints (aka "You've Gotta Start Somewhere"):
  - We will only implement a subset of the functionality that the Stanford Vector provides.
  - `OurVector` will **only store integers** and will not be configurable to store other types
    - Generic, or "templated" classes that allow the client to specify the data type that is stored, are possible in C++, but they are beyond the scope of this class.

What is `OurVector`?

- Goal: Let's make our very own version of the Stanford C++ Vector that we've been using all quarter long.
  - It all will feel so much cooler when we've built it ourselves!

- Scope Constraints (aka "You've Gotta Start Somewhere"):
  - We will only implement a subset of the functionality that the Stanford Vector provides.
  - `OurVector` will **only store integers** and will not be configurable to store other types
    - Generic, or "templated" classes that allow the client to specify the data type that is stored, are possible in C++, but they are beyond the scope of this class.
  - At first, `OurVector` will be limited to **storing a fixed number of elements**, but we will lift this restriction by the end of class. For now, if we run out space we'll just throw an error.

How do we design a class?

We must specify the 3 parts:

1. Member functions: *What functions can you call on a variable of this type?*

2. Member variables: *What subvariables make up this new variable type?*

3. Constructor: *What happens when you make a new instance of this type?*

How do we design `OurVector`?

We must answer the following three questions:

1. Member functions: *What public interface should* **`OurVector`** *support? What functions might a client want to call?*

2. Member variables: *What private information will we need to store in order to keep track of the data stored in* **`OurVector`***?*

3. Constructor: *How are the member variables initialized when a new instance of* **`OurVector`** *is created?*

How do we design `OurVector`?

We must answer the following three questions:

1. **Member functions: *What public interface should* `OurVector` *support? What functions might a client want to call?***

2. Member variables: *What private information will we need to store in order to keep track of the data stored in* **`OurVector`***?*

3. Constructor: *How are the member variables initialized when a new instance of* **`OurVector`** *is created?*

`OurVector` Public Interface

```
class OurVector {
public:
    OurVector();

    void add(int value);
    void insert(int index, int value);
    int get(int index);
    void remove(int index);
    int size();
    bool isEmpty();

private:
    /* To be defined soon! */
};
```

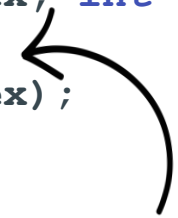
`OurVector` Public Interface

```
class OurVector {
public:
    OurVector();

    void add(int value);
    void insert(int index, int value);
    int get(int index);
    void remove(int index);
    int size();
    bool isEmpty();

private:
    /* To be defined soon! */
};
```

These methods should all look very familiar – we've been using them all quarter long!

`OurVector` Public Interface

```
class OurVector {
public:
    OurVector();

    void add(int value);
    void insert(int index, int value);
    int get(int index);
    void remove(int index);
    int size();
    bool isEmpty();

private:
    /* To be defined soon! */
};
```



We'll use the get method to emulate the functionality of the [] operator.

`OurVector` Public Interface

```
class OurVector {
public:
    OurVector();

    void add(int value);
    void insert(int index, int value);
    int get(int index);
    void remove(int index);
    int size();
    bool isEmpty();

private:
    /* To be defined soon! */
};
```

What should go here?

How do we design **`OurVector`**?

We must answer the following three questions:

1. Member functions: *What public interface should* **`OurVector`** *support? What functions might a client want to call?*

2. **Member variables: *What private information will we need to store in order to keep track of the data stored in* `OurVector`*?***

3. Constructor: *How are the member variables initialized when a new instance of* **`OurVector`** *is created?*

OurVector Member Variables

OurVector Member Variables

- `int* elements;`
  - A pointer to an array of integers, which will act as our underlying data storage mechanism.

OurVector Member Variables

- `int* elements;`
  - A pointer to an array of integers, which will act as our underlying data storage mechanism.

- `int allocatedCapacity;`
  - An integer that stores the size of the allocated elements array. Remember, arrays don't have any conception/knowledge of their own size, so we must manually track this!

OurVector Member Variables

- `int* elements;`
  - A pointer to an array of integers, which will act as our underlying data storage mechanism.

- `int allocatedCapacity;`
  - An integer that stores the size of the allocated elements array. Remember, arrays don't have any conception/knowledge of their own size, so we must manually track this!

- `int numItems;`
  - An integer that stores the number of elements currently stored in the vector.

`OurVector` Header File

```
class OurVector {
public:
    OurVector();

    void add(int value);
    void insert(int index, int value);
    int get(int index);
    void remove(int index);
    int size();
    bool isEmpty();
private:
    int* elements;
    int allocatedCapacity;
    int numItems;
};
```

How do we design `OurVector`?

We must answer the following three questions:

1. Member functions: *What public interface should* **`OurVector`** *support? What functions might a client want to call?*

2. Member variables: *What private information will we need to store in order to keep track of the data stored in* **`OurVector`***?*

3. **Constructor: *How are the member variables initialized when a new instance of* `OurVector` *is created?***

Review: Constructors

- A constructor is a special member function used to set up the class before it is used.
- The constructor is automatically called when the object is created.
- The constructor for a class named **`ClassName`** has signature **`ClassName(args);`**

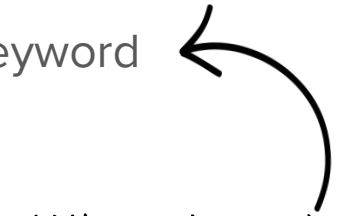
```
class OurVector {
public:
    OurVector();

    void add(int value);
    void insert(int index, int value);
    int get(int index);
    void remove(int index);
    int size();
    bool isEmpty();
private:
    int* elements;
    int allocatedCapacity;
    int numItems;
};
```

`OurVector` Constructor

- The constructor must initialize all the values of our member variables to be things that initially make sense
- The **`allocatedCapacity`** should be set to some small integer
- The **`elements`** array should be allocated using the **`new[]`** keyword
- The **`numItems`** counter should be initialized to 0

`OurVector` Constructor

- The constructor must initialize all the values of our member variables to be things that initially make sense
- The **`allocatedCapacity`** should be set to some small integer
- The **`elements`** array should be allocated using the **`new[]`** keyword
- The **`numItems`** counter should be initialized to 0



Destructors

- A destructor is a special member function responsible for cleaning up an object's memory.
- It’s automatically called whenever an object’s lifetime ends (for example, if it’s a local variable that goes out of scope).
- The destructor for a class named **`ClassName`** has signature **`~ClassName();`**

Destructors

- A destructor is a special member function responsible for cleaning up an object's memory.
- It's automatically called whenever an object's lifetime ends (for example, if it's a local variable that goes out of scope).
- The destructor for a class named **`ClassName`** has signature **`~ClassName();`**

```
class OurVector {
public:
    OurVector();

    void add(int value);
    void insert(int index, int value);
    int get(int index);
    void remove(int index);
    int size();
    bool isEmpty();
private:
    int* elements;
    int allocatedCapacity;
    int numItems;
};
```

Destructors

- A destructor is a special member function responsible for cleaning up an object's memory.
- It's automatically called whenever an object's lifetime ends (for example, if it's a local variable that goes out of scope).
- The destructor for a class named **`ClassName`** has signature **`~ClassName();`**

```
class OurVector {
public:
    OurVector();
    ~OurVector();
    void add(int value);
    void insert(int index, int value);
    int get(int index);
    void remove(int index);
    int size();
    bool isEmpty();
private:
    int* elements;
    int allocatedCapacity;
    int numItems;
};
```

`OurVector` Destructor

- The destructor must take responsibility for freeing any allocated memory currently in use by an instance of the class.

- In particular, this means calling the **`delete[]`** operator on the elements array to officially give that memory back to the computer and avoid any memory leaks.

- The other member variables (**`allocatedCapacity`** and **`numItems`**) are both simple stack-allocated variables, so nothing special is needed to clean them up.

Let's Code It! (Part 1)

Member Variables, Constructor, and Destructor

Summary

- Member variables define the key data storage components of a class implementation.

- The **constructor** is the special method that gets called when a new instance of a class is declared. In this method, we initialize all of our member variables to the appropriate values, including allocating any necessary memory.

- The **destructor** is a special method that gets called when an instance of a class goes out of scope and thus is destroyed. In this method, we most often are responsible for freeing any dynamically allocated memory used by the instance.

Announcements

Announcements

- Midterm feedback/grades and solutions were released on Monday 11:55 PM. Please see our post on Ed for all the information.

- Assignment 3 is due **tonight at 11:59pm PDT**. Assignment 4 will be released by the end of the day today and will be due **next Tuesday**.

- Your project proposal is due **this Sunday** so get started soon.

Final Project

For your capstone final project, you will:

- Pick a topic from this quarter that targets a potential area of growth and write your own problem on that topic (scoped for a section/diagnostic/exam). Write a project proposal explaining your project idea. Due **Sunday, July 24**.

- Write a project report that includes a problem description, multiple possible solutions, and a capstone ethics reflection. Due **Sunday, August 7**.

- Take on the role of a section leader and teach/present the problem to your section leader in a 30-minute, 1-on-1 session. **Takes place from August 11-14**.

Visualizing `OurVector` Operations

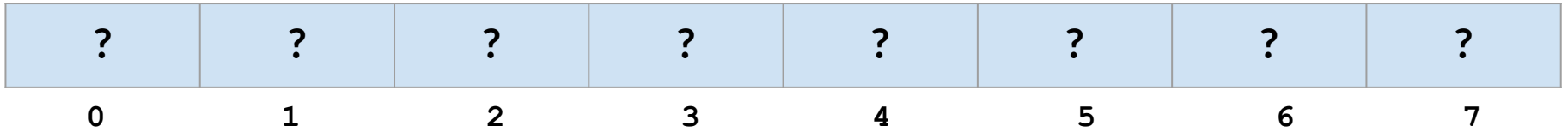
Initialization

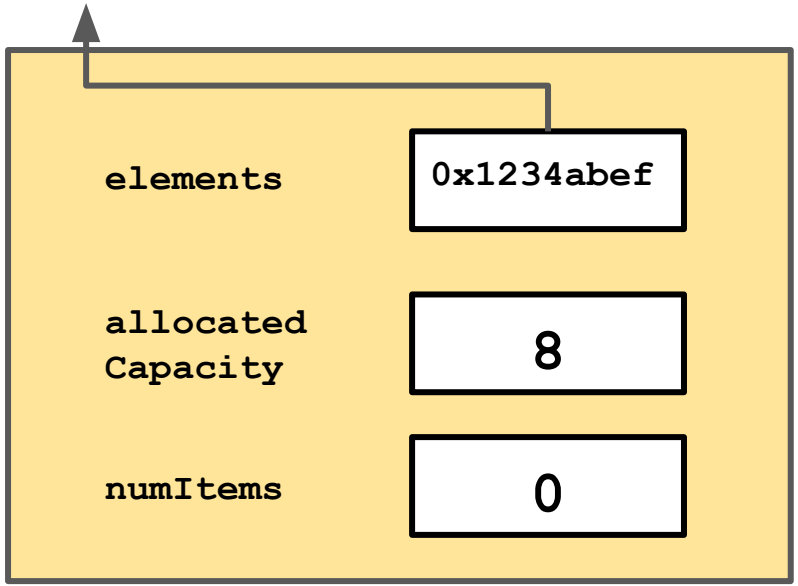
Initialization via the Constructor

Initialization via the Constructor

```
// client code

OurVector vec;
```

Initialization via the Constructor

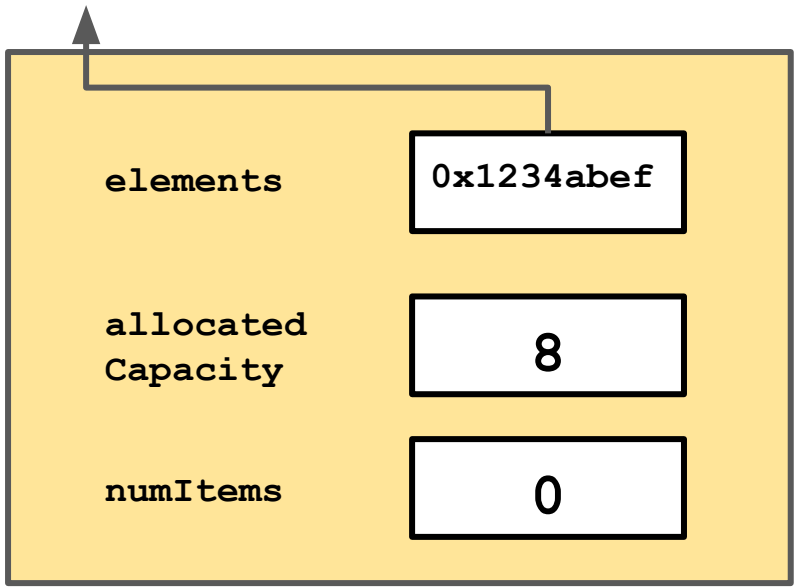





```
// client code

OurVector vec;
```

Initialization via the Constructor

Newly allocated arrays initially store random (or garbage) values



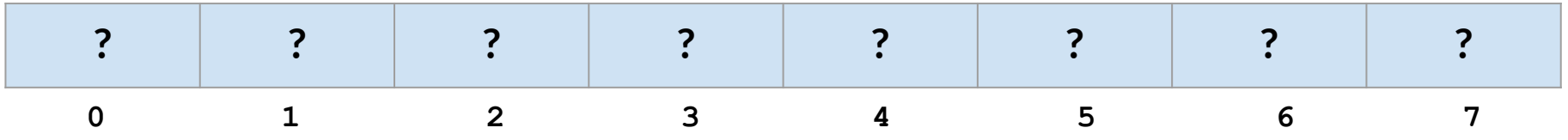

```
// client code

OurVector vec;
```

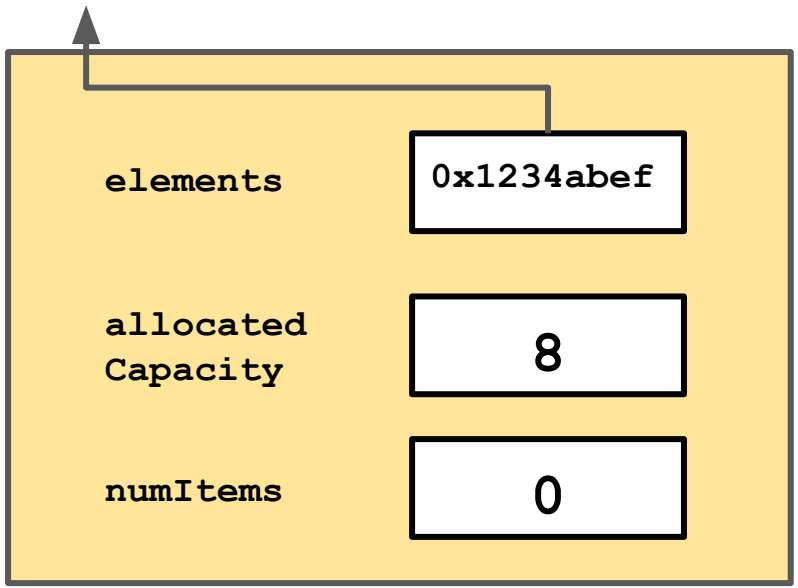
Adding Elements

The `add()` operation

- The `add()` operation is responsible for taking a specified element and adding it to the first open spot at the end of the vector.

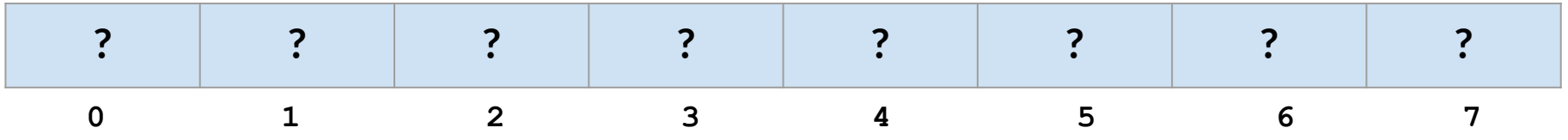
The `add()` operation

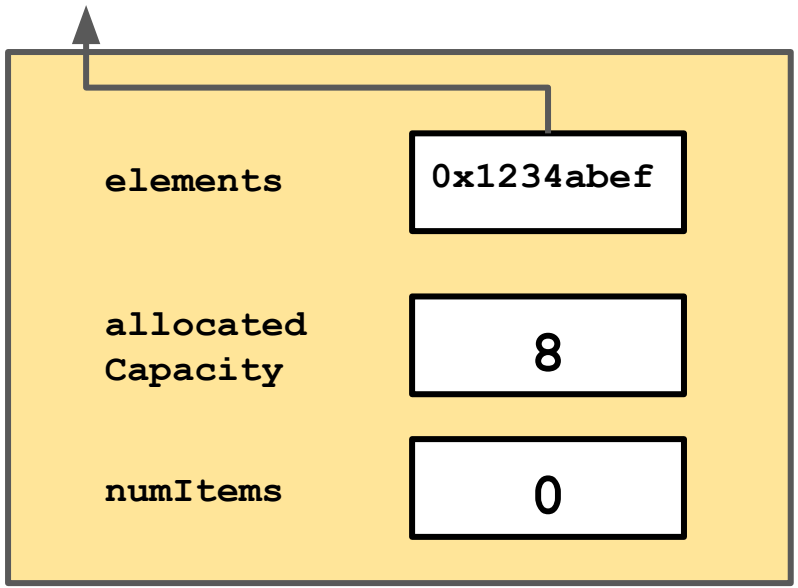





```
// client code

OurVector vec;
```

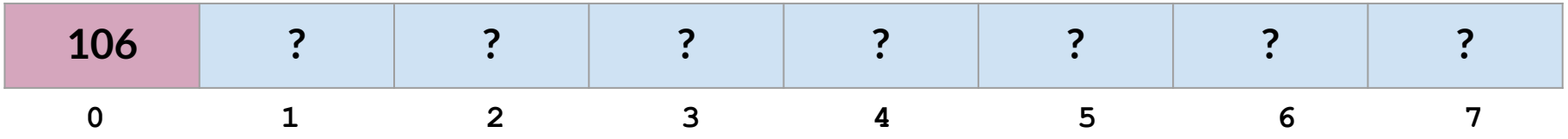
The `add()` operation

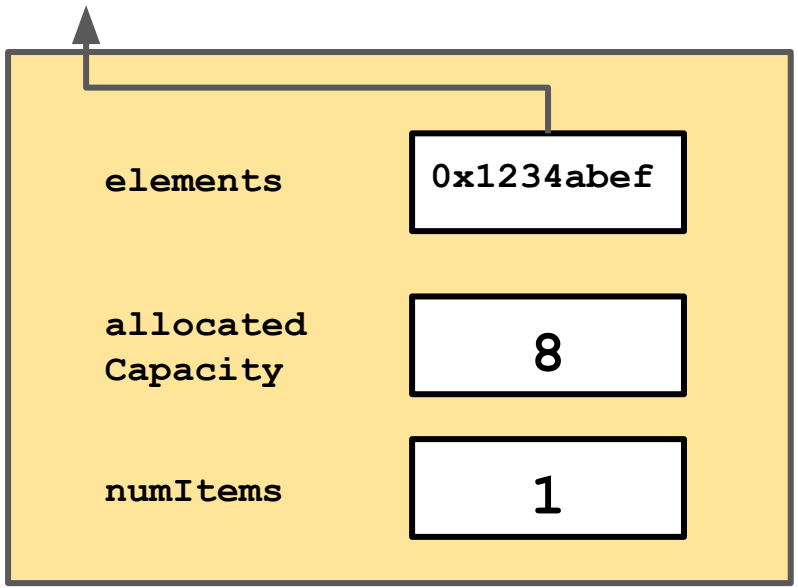





```
// client code

OurVector vec;
vec.add(106);
```

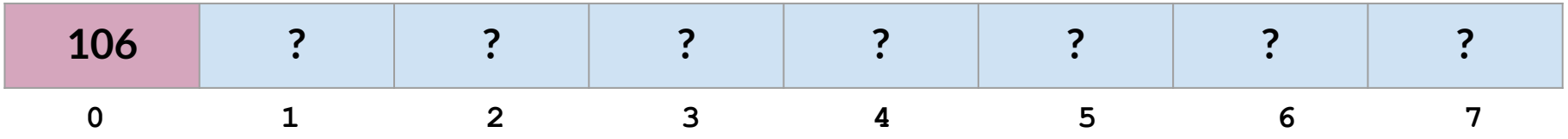
The `add()` operation

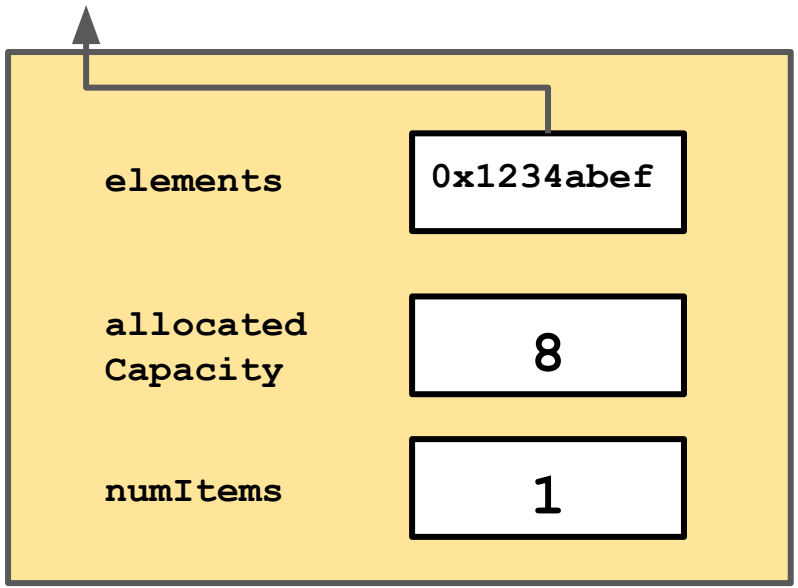





```
// client code

OurVector vec;
vec.add(106);
```

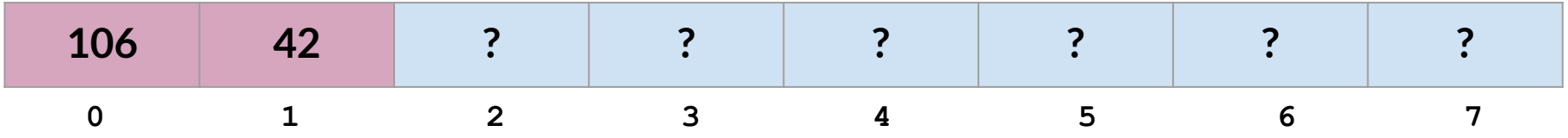
The `add()` operation

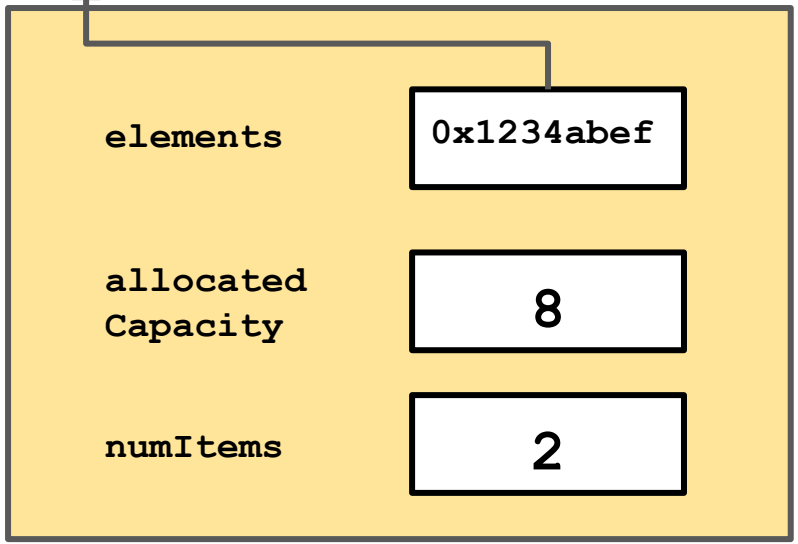





```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
```

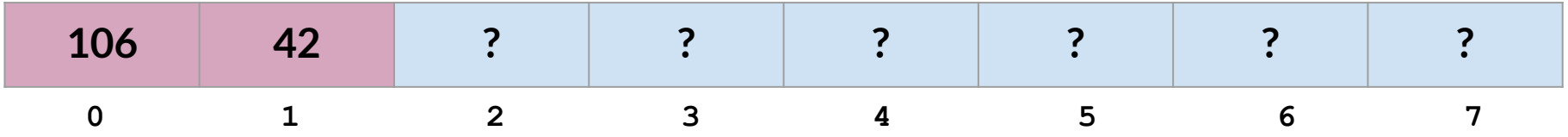
The `add()` operation

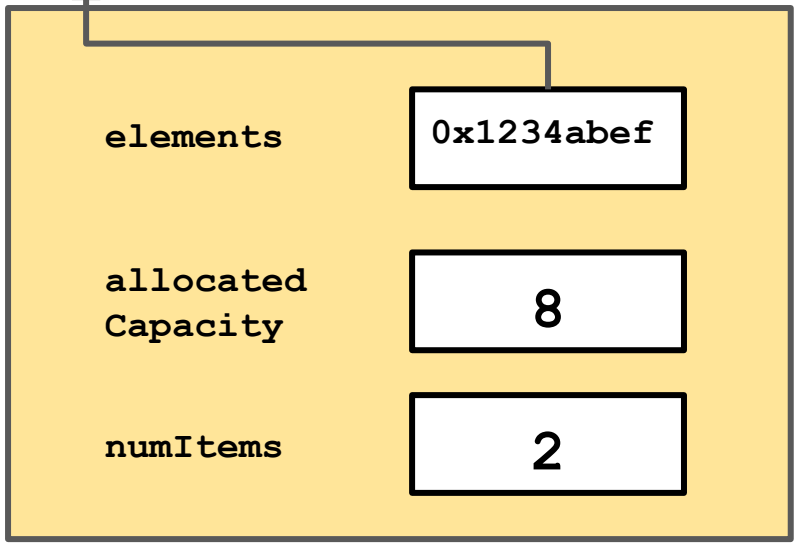





```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
```

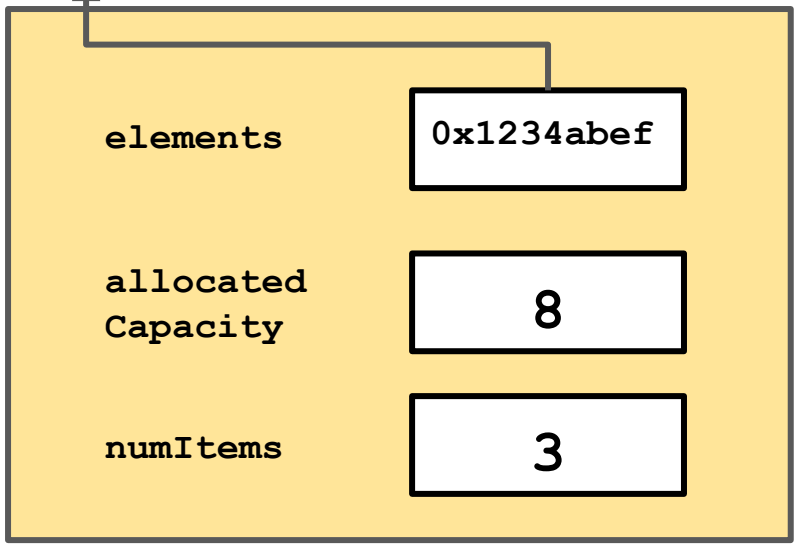
The `add()` operation





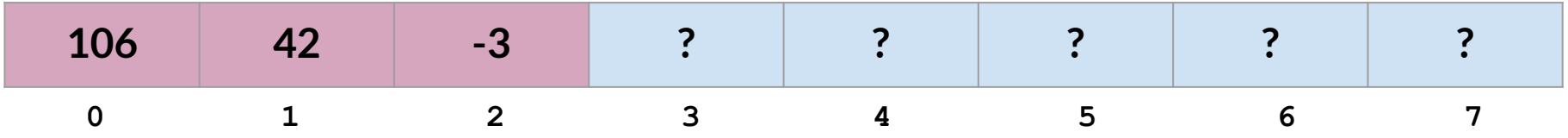

```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
```

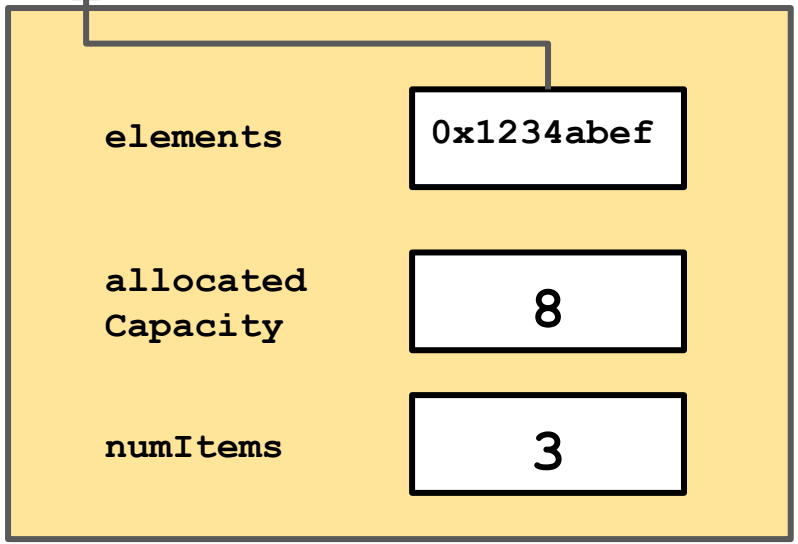
The `add()` operation




```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
```

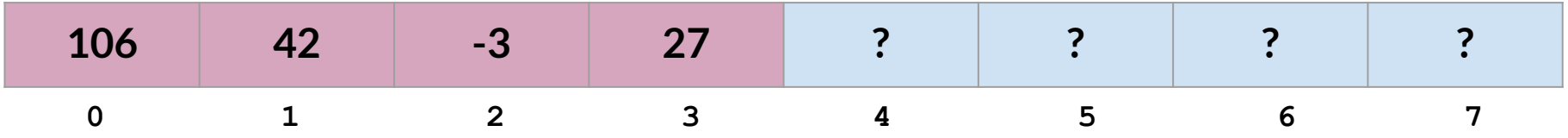
The `add()` operation

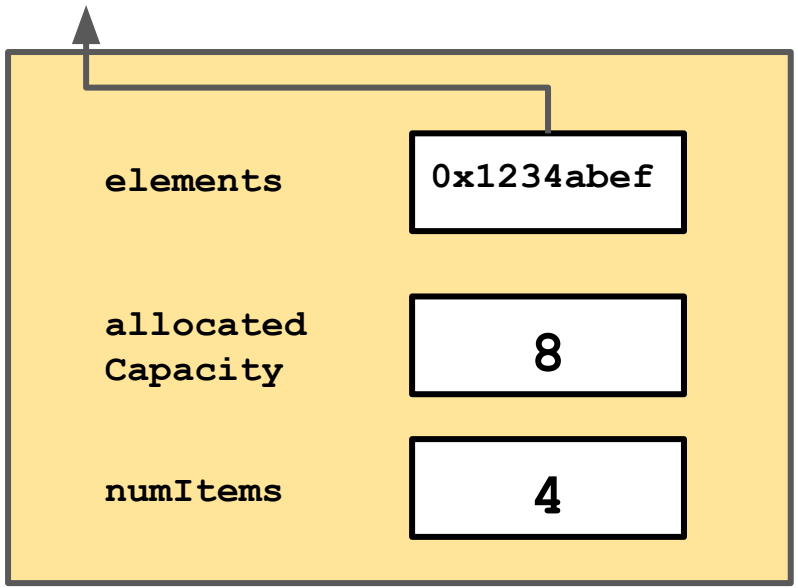





```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);
```

The `add()` operation





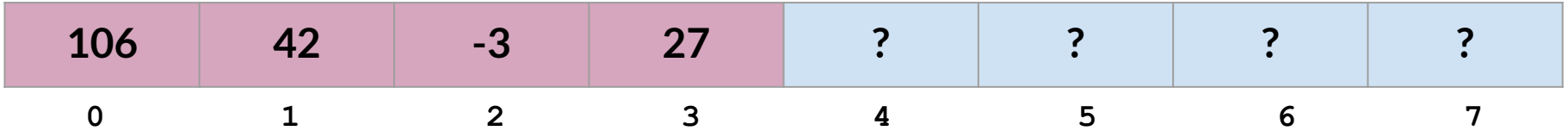

```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);
```

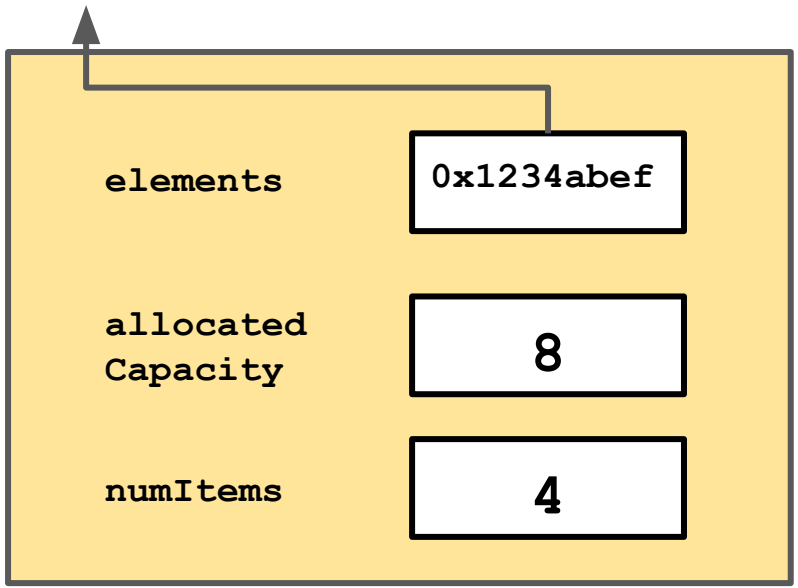
Removing Elements

The `remove()` operation

- The `remove()` operation allows the client to specify an index at which to remove an element, and then removes the value at that index.

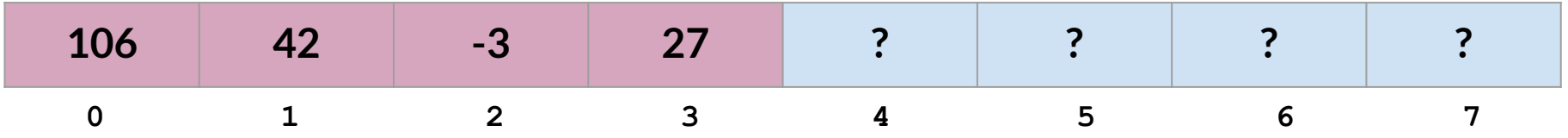
The `remove()` operation

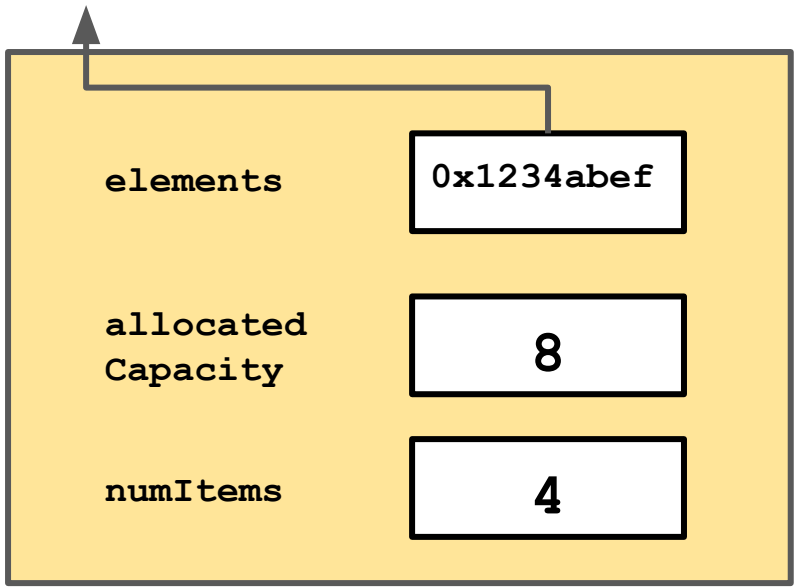





```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);
```

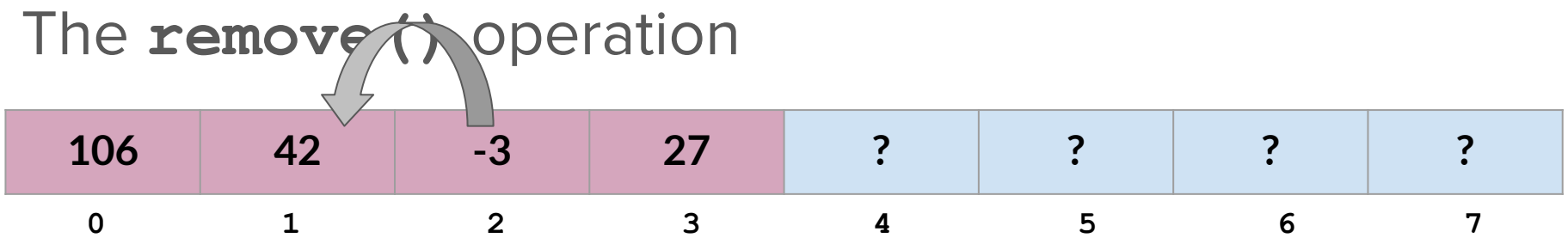
The `remove()` operation





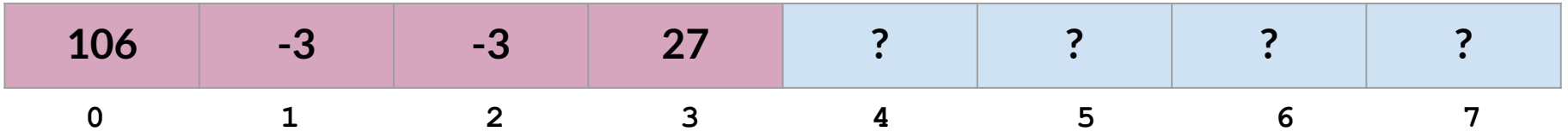

```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

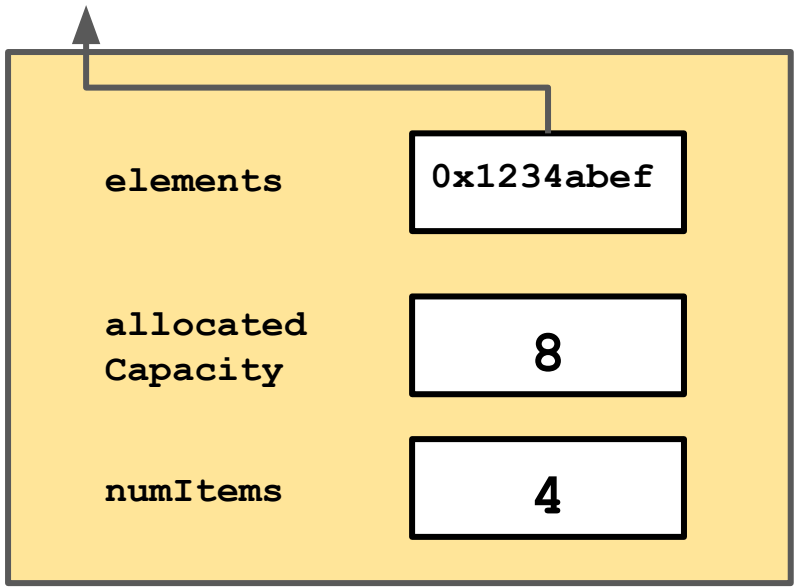
The `remove()` operation




```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

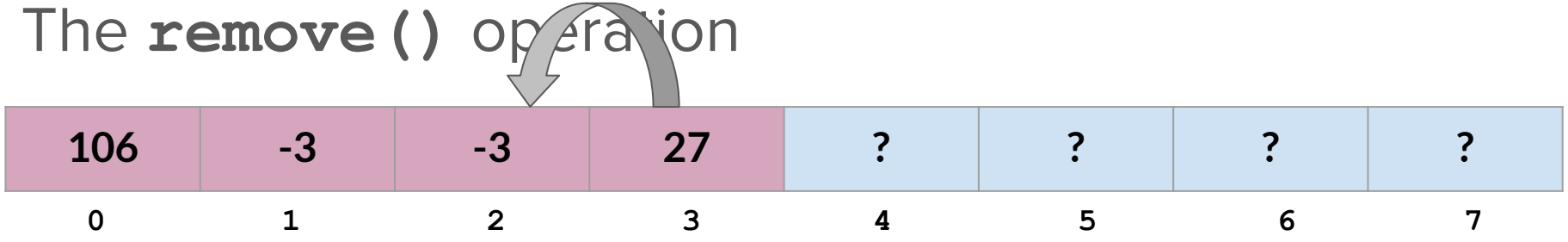
The `remove()` operation

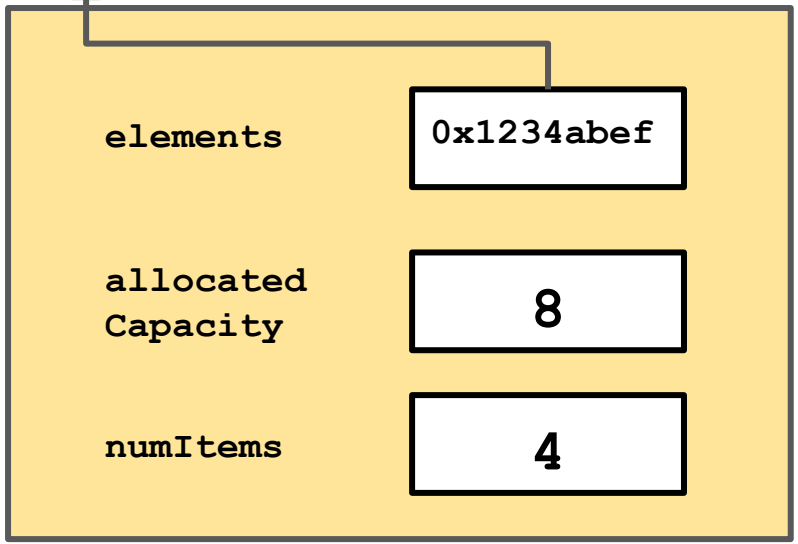





```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

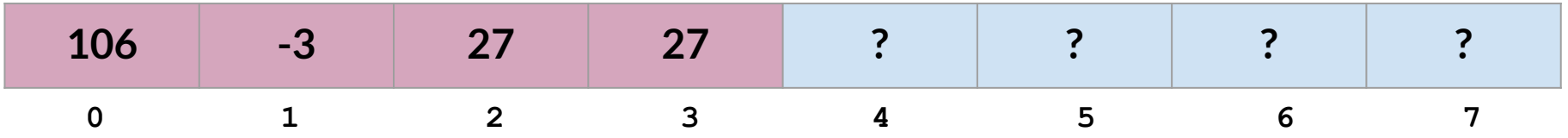
The `remove()` operation

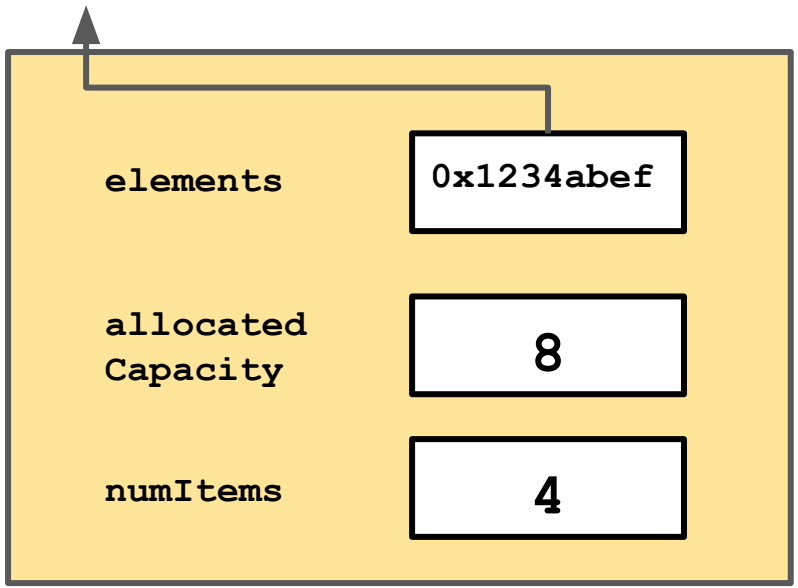





```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

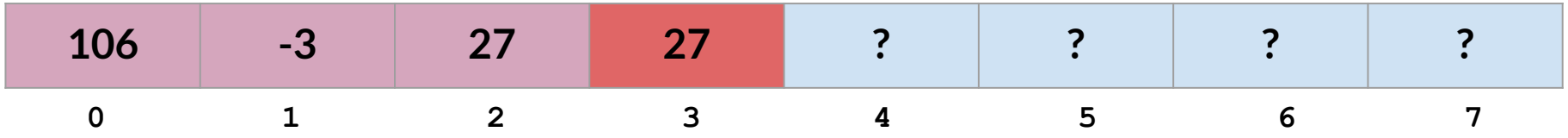
The `remove()` operation

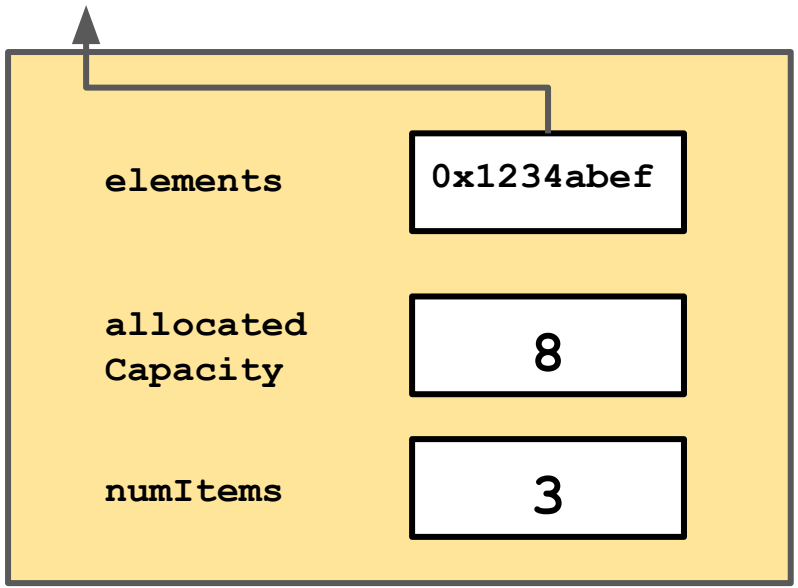





```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

The `remove()` operation

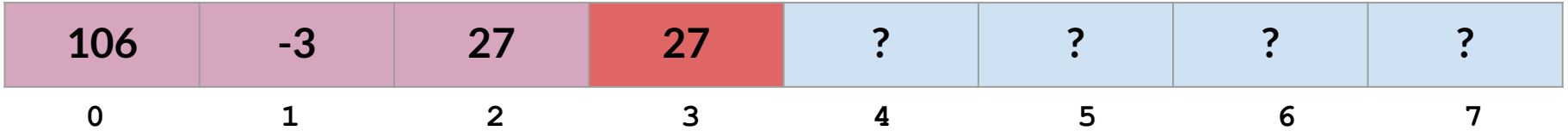


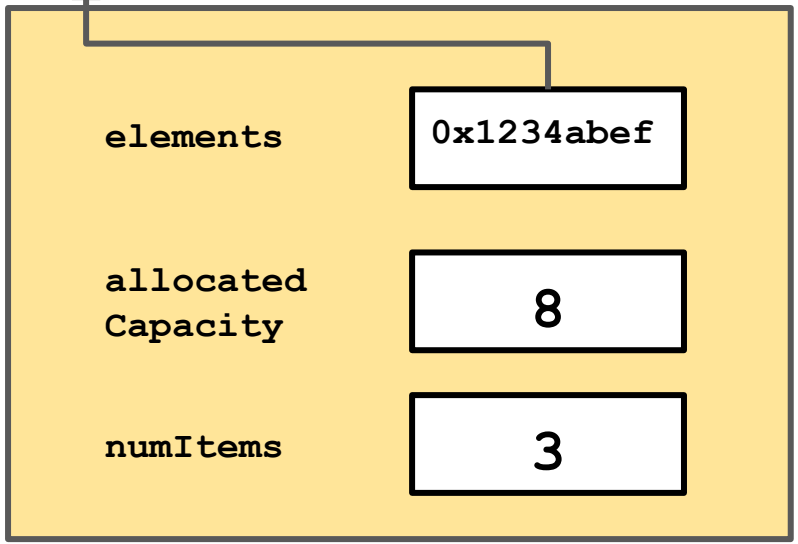



```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

The `remove()` operation

Arrays cannot grow or shrink, so this older value is still technically there in the array. We're just going to pretend that it isn't!

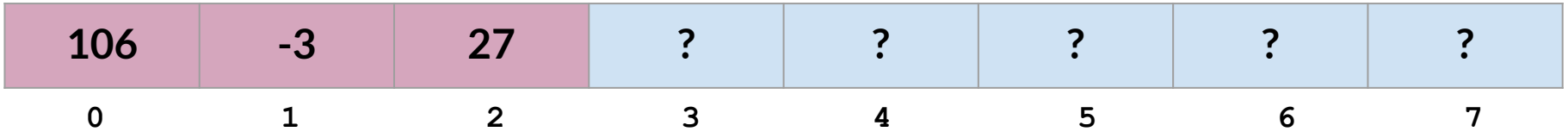


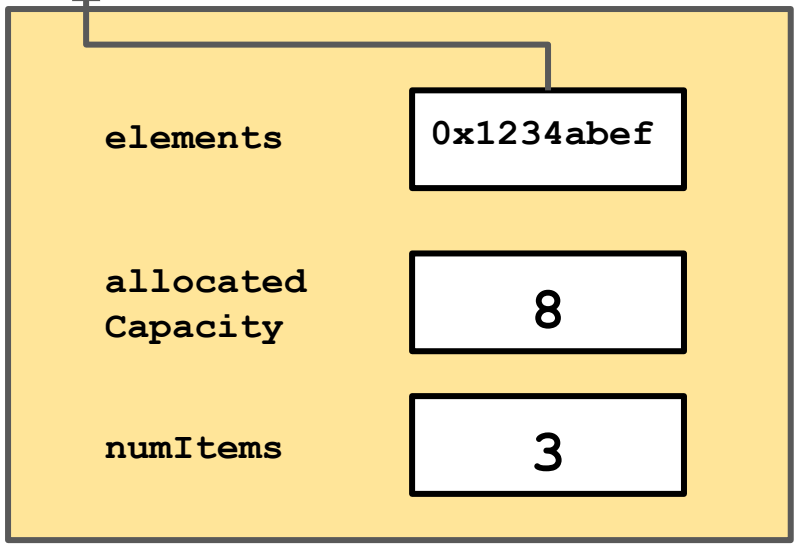



```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

The `remove()` operation

Arrays cannot grow or shrink, so this older value is still technically there in the array. We're just going to pretend that it isn't!





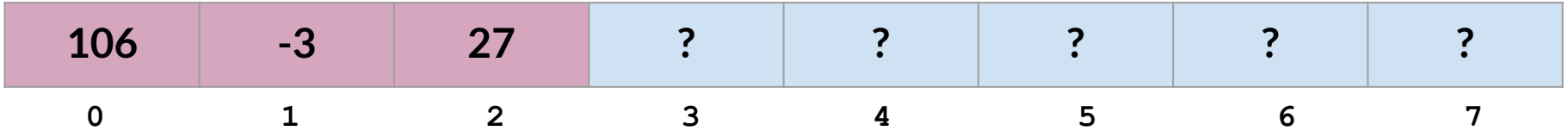

```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

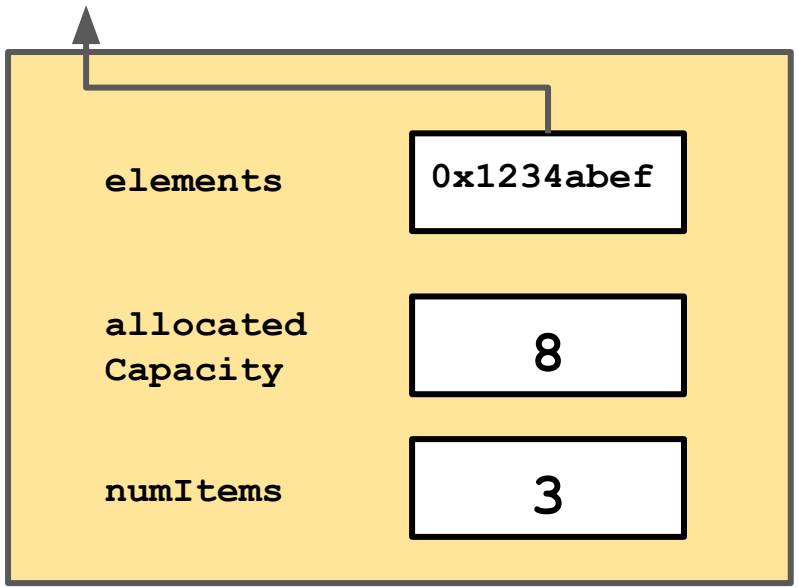
Inserting Elements

The **`insert()`** operation

- The **`insert()`** operation is similar to **`add()`**, but allows the client to specify which index they want the value to be inserted at.

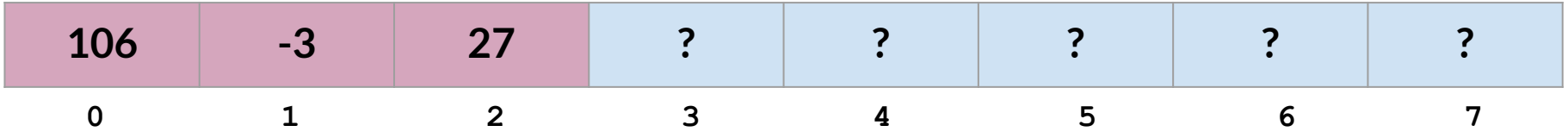
The `insert()` operation

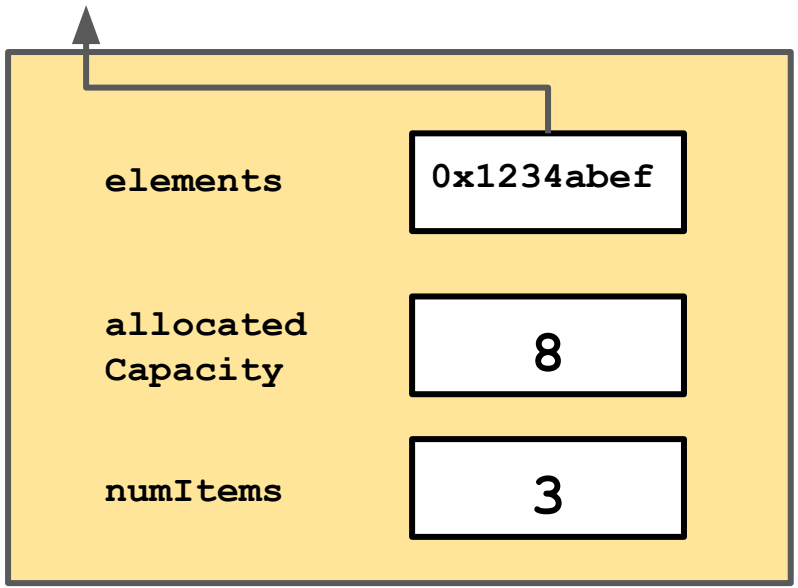





```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
```

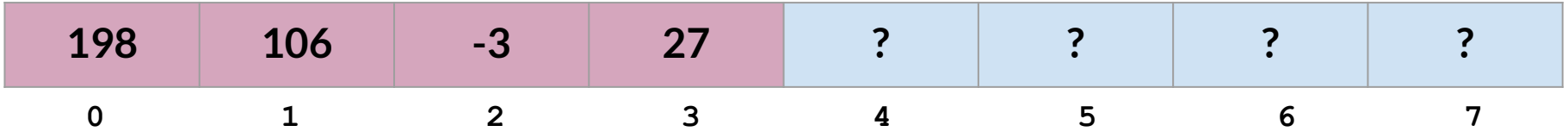
The `insert()` operation

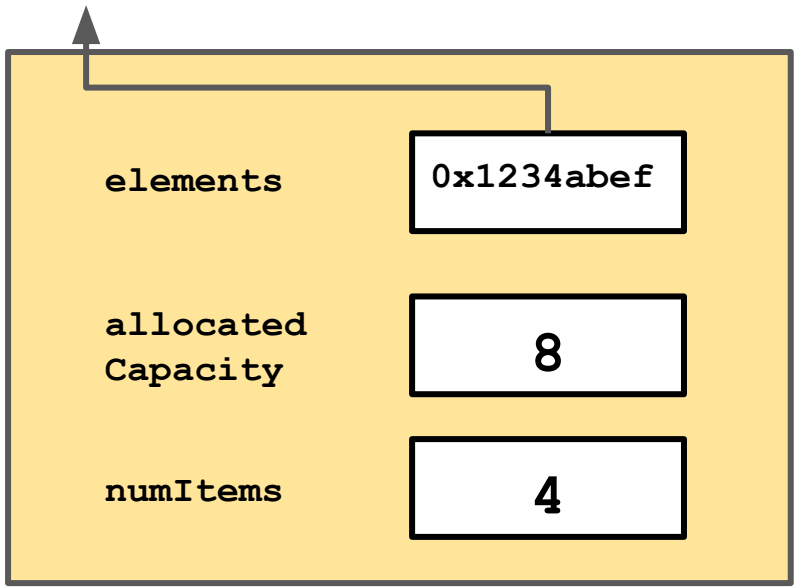





```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
vec.insert(0, 198);
```

The `insert()` operation






```
// client code

OurVector vec;
vec.add(106);
vec.add(42);
vec.add(-3);
vec.add(27);

vec.remove(1);
vec.insert(0, 198);
```

Assorted Operations

The `get()` / `size()` / `isEmpty()` operations

- The remaining operations that we have left to implement should be relatively straightforward, given the member variables we have.
  - `int* elements`
  - `int allocatedCapacity`
  - `int numItems`

- The `get()` method should return the array element at the specified index.
- The `size()` method should return the number of items the array currently holds (not the total allocated capacity).
- The `isEmpty()` method return true if number of items is > 0.

Attendance ticket:

https://tinyurl.com/stanfordvector

Please don’t send this link to students who are not here. It’s on your honor!

The `get()` / `size()` / `isEmpty()` operations

- The remaining operations that we have left to implement should be relatively straightforward, given the member variables we have.
- The `get()` method can just return the array element at the specified index.
- The `size()` method can just return the value of the `numItems` member variable.
- The `isEmpty()` method can compare `numItems` to 0 and return the appropriate result.

Implementing `OurVector`

Let's Code It! (Part 2)

`add()`, `remove()`, `insert()`, `get()`, `size()`, `isEmpty()`

Summary

- Using an array as a backing store of data involves shifting elements around – this kind of code is ripe for off-by-one errors!

- With good member variable member choices, most public methods are relatively straightforward to implements.

- We've now gained an appreciation for *why* insertion/removal on Vectors is an "expensive" O(n) operation.

Running Out of Space

- Our current implementation very quickly runs out of space to store elements.

- What should we do when this happens?
  - Currently, we just throw an error. That doesn't seem quite right. What if all data structures we used were limited to hold only 8 items?
  - Instead, we need a way to **dynamically resize (grow)** our internal data storage mechanism.

Dynamic Array Growth

A Day in the Life of a Hermit Crab

- Hermit crabs are interesting animals. They live in scavenged shells that they find on the seafloor. Once in a shell, this is their lifestyle (with a bit of poetic license):

A Day in the Life of a Hermit Crab

- Hermit crabs are interesting animals. They live in scavenged shells that they find on the seafloor. Once in a shell, this is their lifestyle (with a bit of poetic license):
  - Grow until they have outgrown their current shell. Then, follow these 5 steps.
    - Find another, larger shell.
    - Move all their stuff into the new shell.
    - Leave the old shell on the seafloor.
    - Update their address with the Hermit Crab Postal Service.
    - Make note of their new shell's spacious capacity by posting on Hermit Crab Instagram.

A Day in the Life of a Hermit Crab

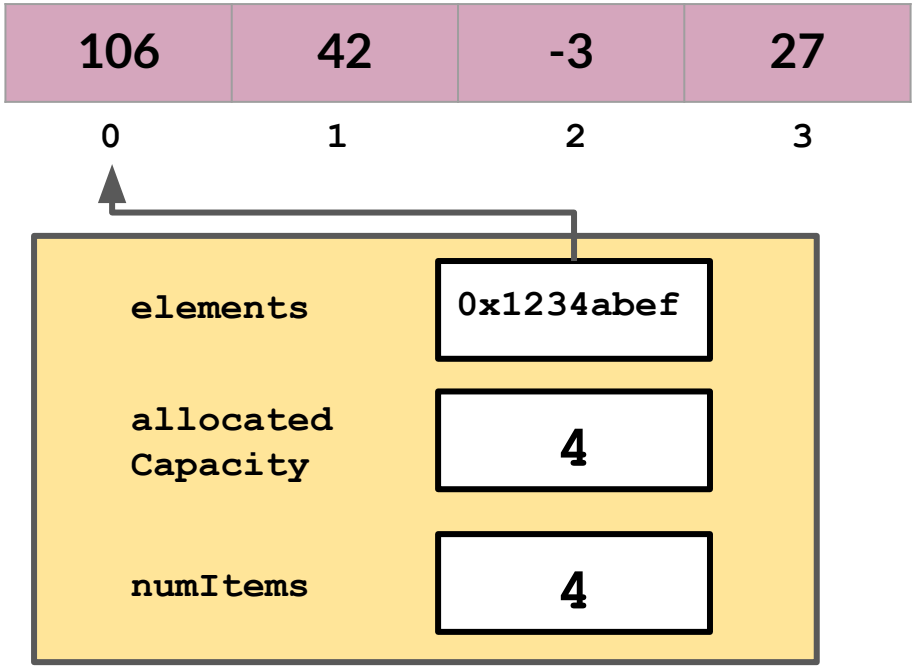
- Hermit crabs are interesting animals. They live in scavenged shells that they find on the seafloor. Once in a shell, this is their lifestyle (with a bit of poetic license):
  - Grow until they have outgrown their current shell. Then, follow these 5 steps.
    - Find another, larger shell.
    - Move all their stuff into the new shell.
    - Leave the old shell on the seafloor.
    - Update their address with the Hermit Crab Postal Service.
    - Make note of their new shell's spacious capacity by posting on Hermit Crab Instagram.
- While this is purposefully a bit of a silly analogy, this process models almost exactly what we need to do in order to dynamically resize our internal data storage mechanism.

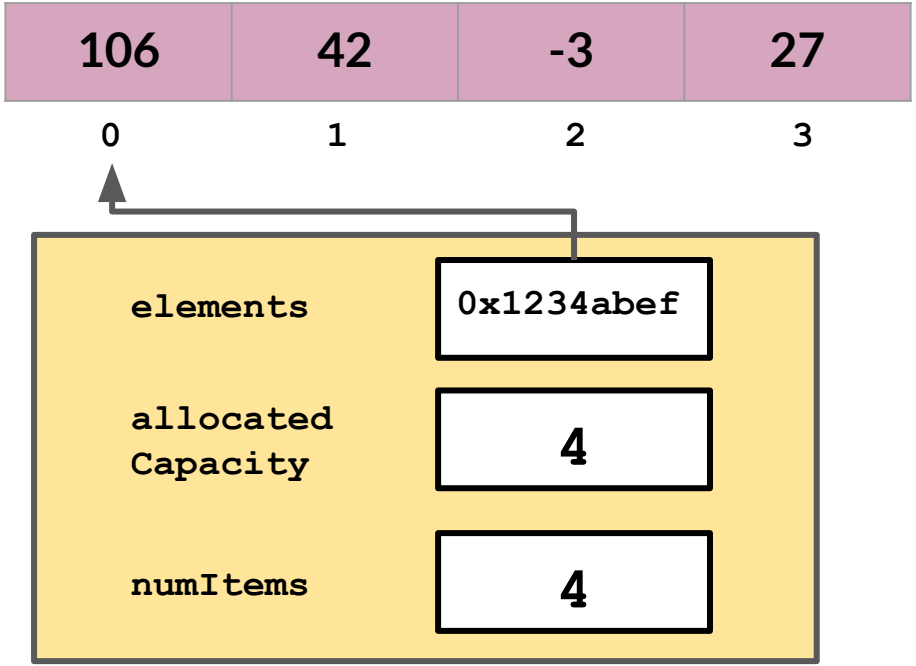
A Day in the Life of a Growable Array

- In essence, when we run out of space in our array, we want to allocate a new array that is bigger than our old array so we can store the new data and keep growing. These "growable arrays" follow a five-step expansion that mirrors the hermit crab model (with poetic license).

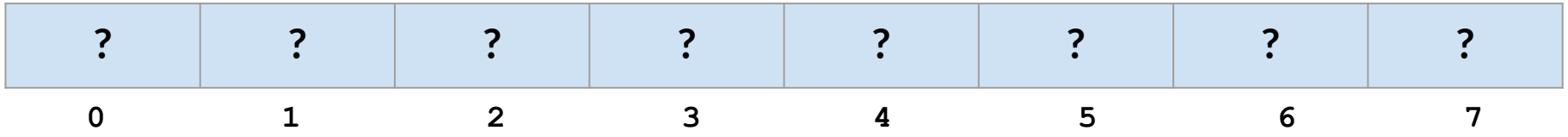
A Day in the Life of a Growable Array

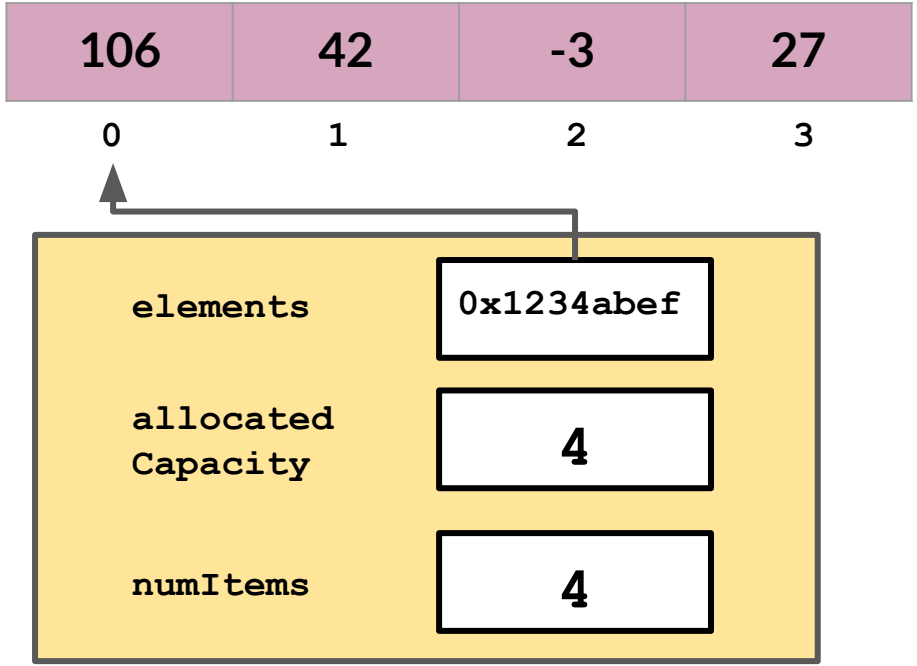
- In essence, when we run out of space in our array, we want to allocate a new array that is bigger than our old array so we can store the new data and keep growing. These "growable arrays" follow a five-step expansion that mirrors the hermit crab model (with poetic license).
  - Grow the array until we run out of space (how can we tell if we've run out of space?)
    - Create a new, larger array. Usually we choose to **double** the current size.
    - Copy the old array elements to the new array.
    - Delete (free) the old array.
    - Point the old array variable to the new array.
    - Update the associated capacity variable for the array.


106
42
-3
27
0
1
2
3
elements
0x1234abef
allocated
Capacity
4
numItems
4

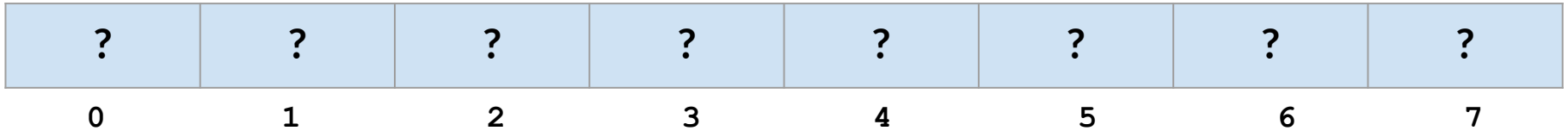


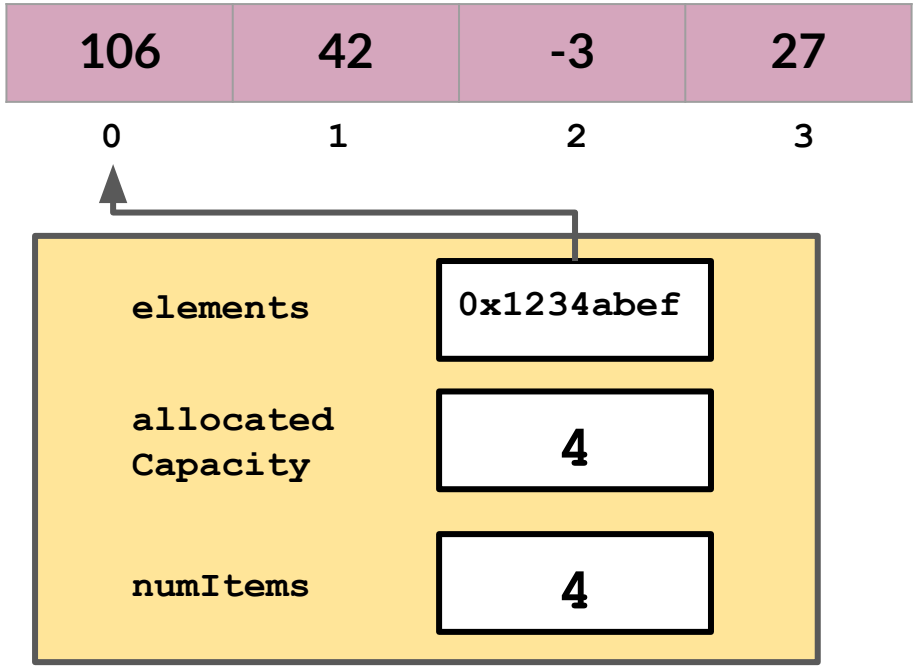
1. Create a new, larger array. Usually we choose to double the current size.



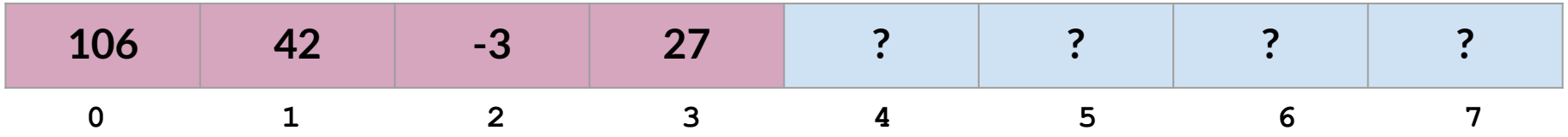


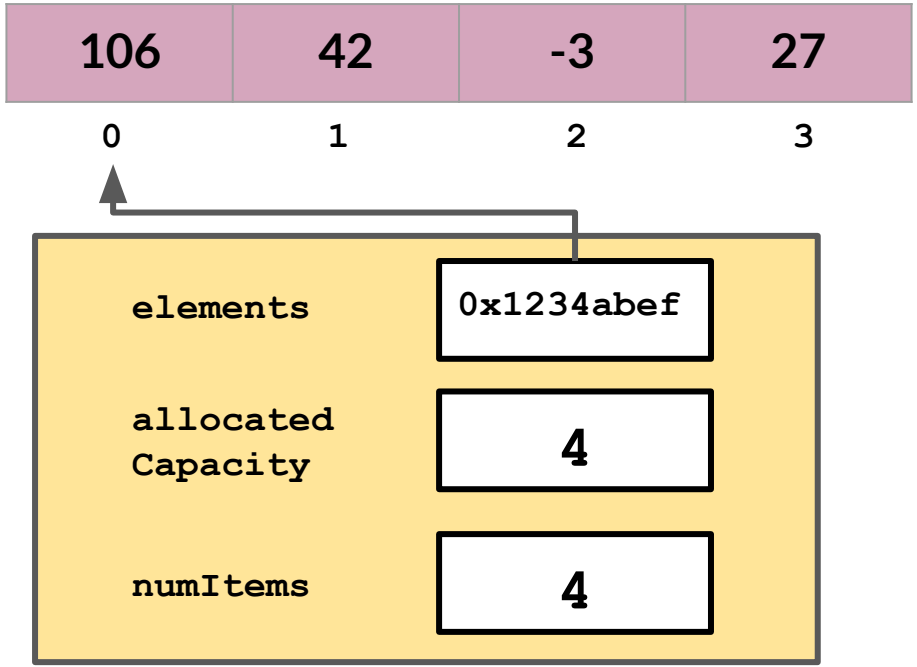
1. Create a new, larger array. Usually we choose to double the current size.



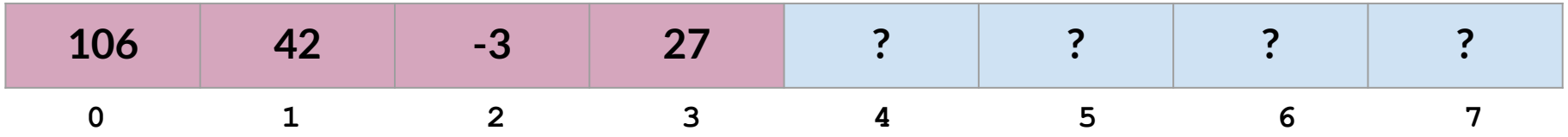


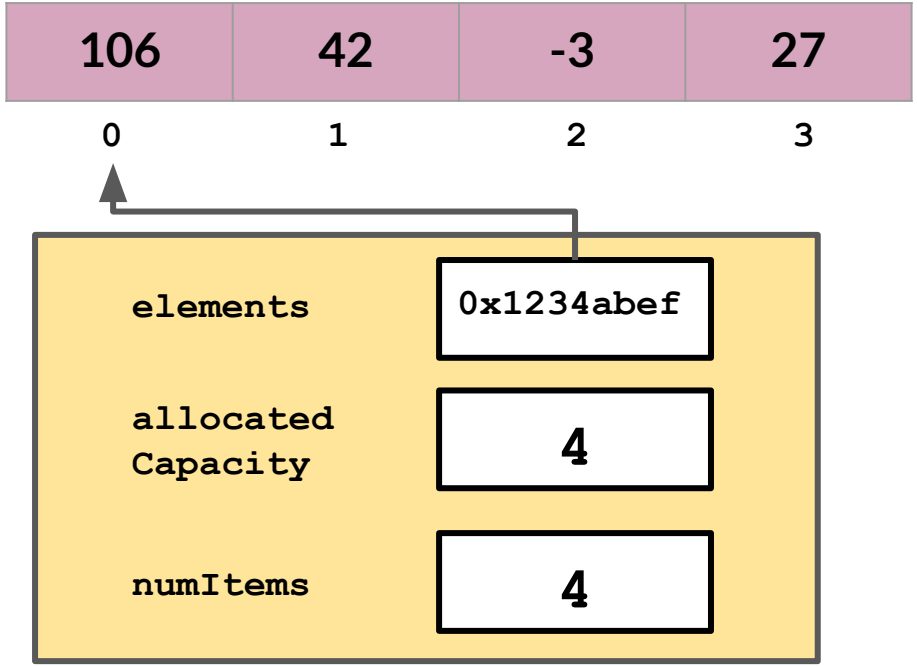
1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.



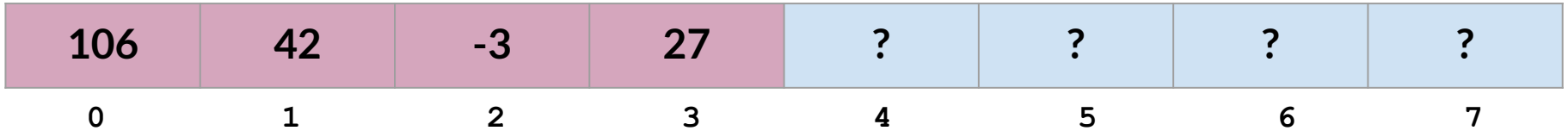


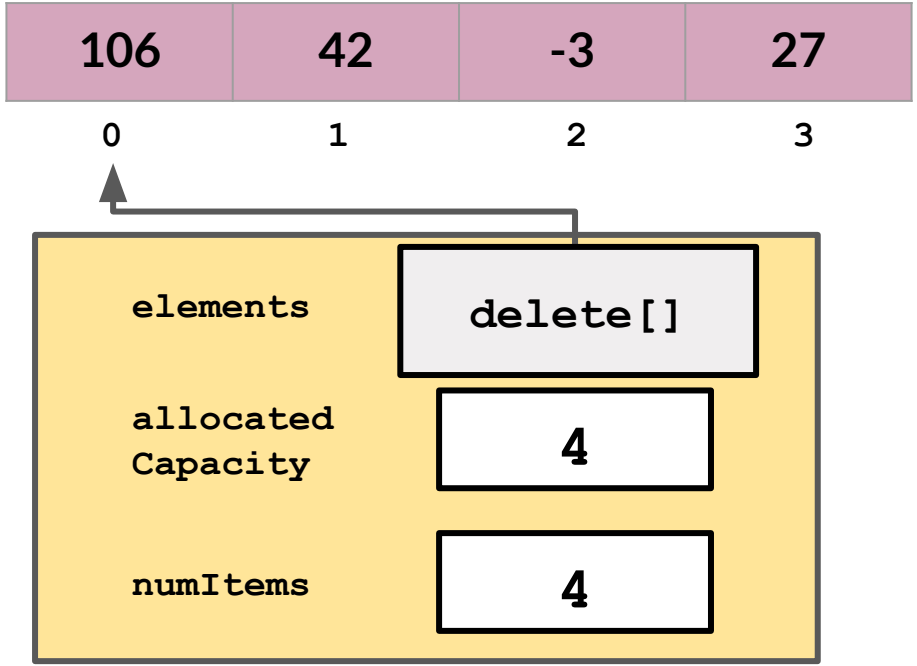
1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.



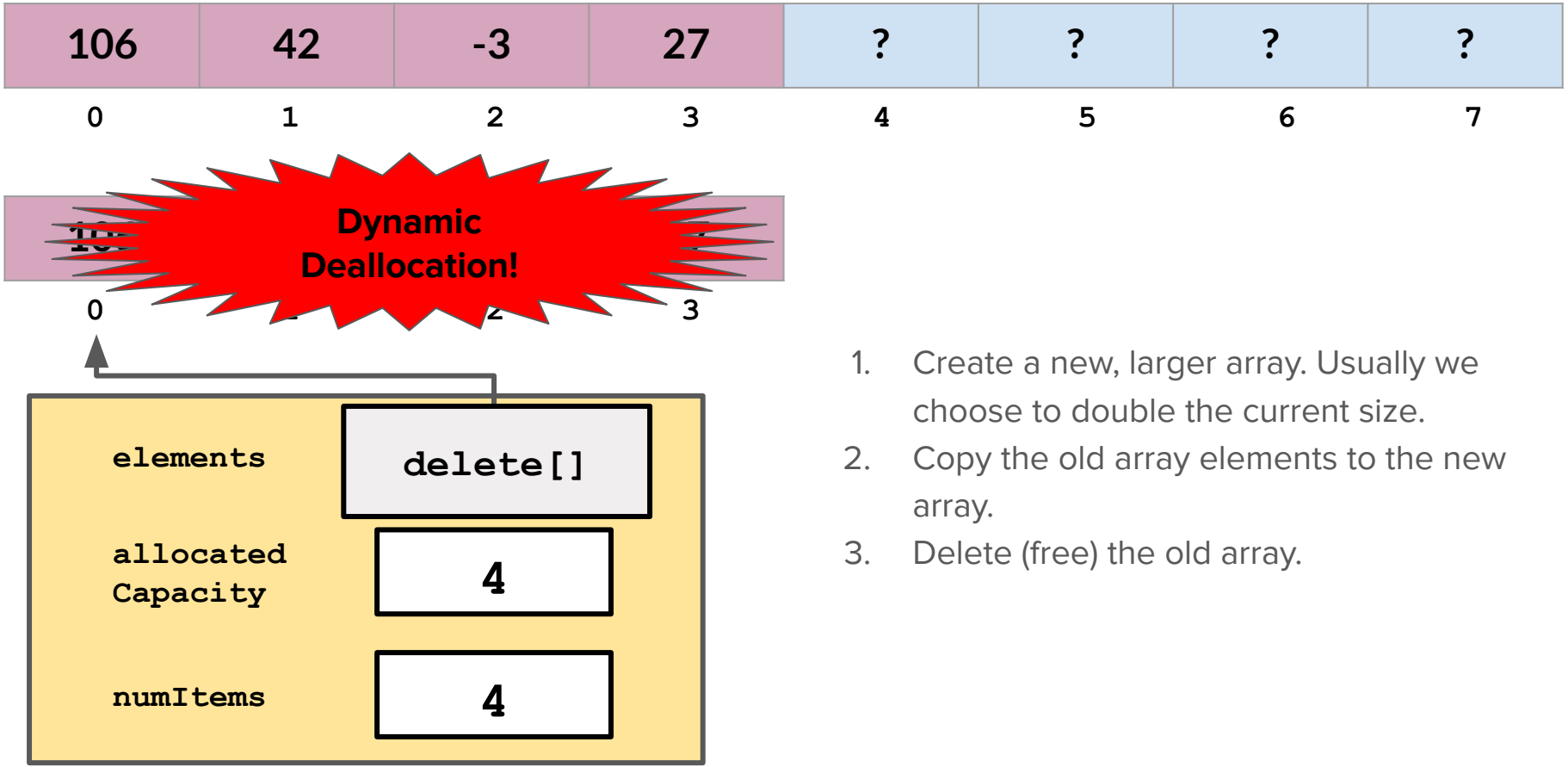


1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.



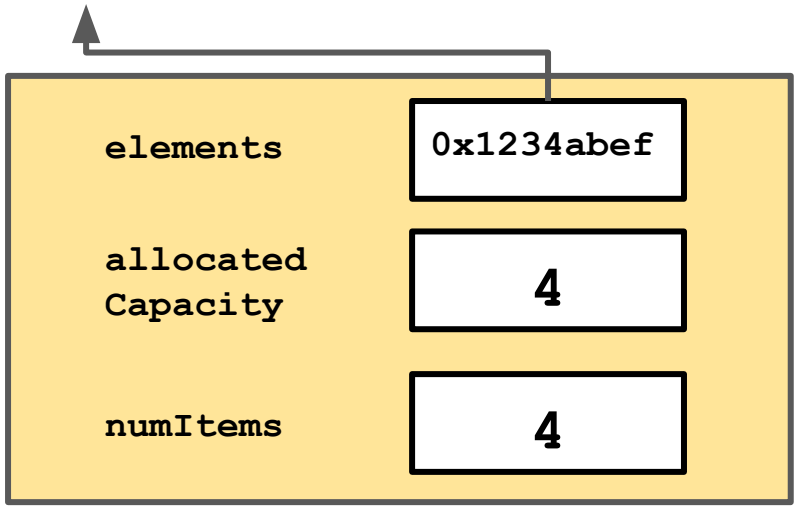


1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.



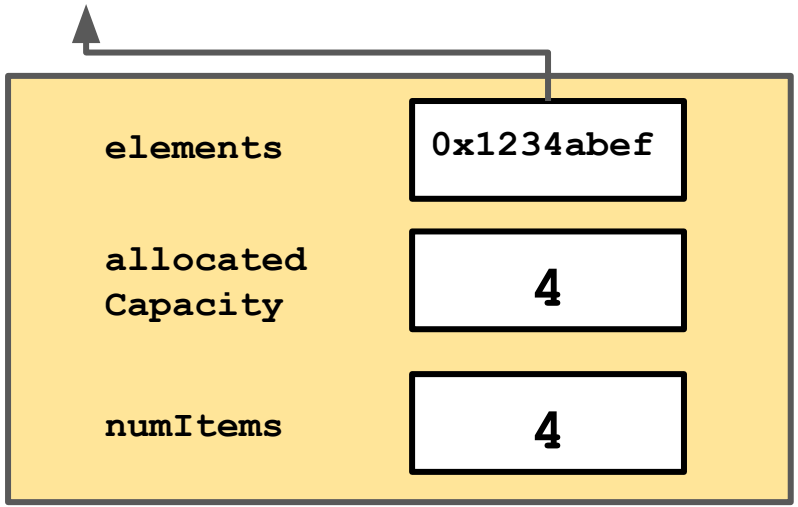
1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.

106	42	-3	27	?	?	?	?
0	1	2	3	4	5	6	7

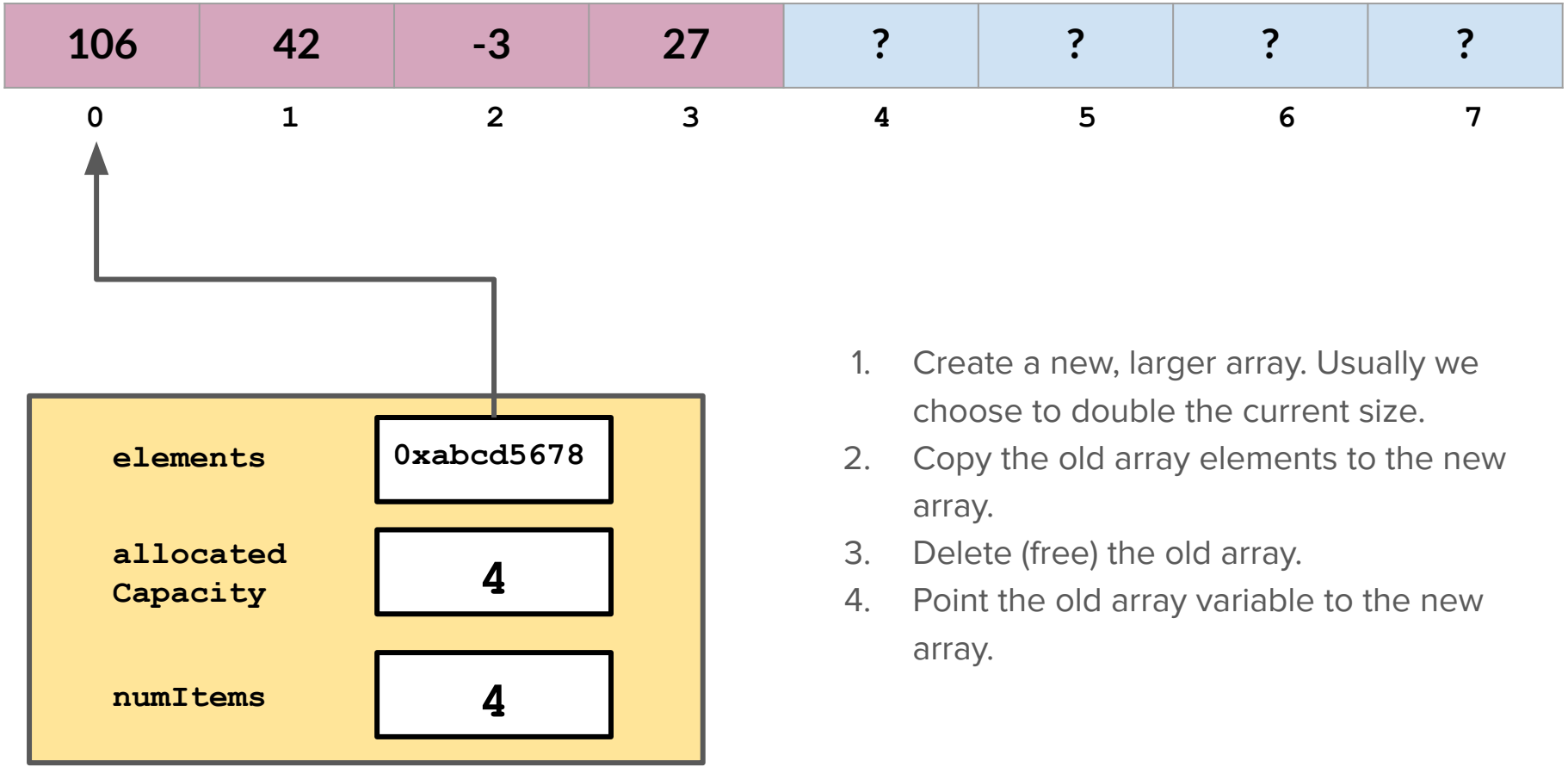


1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.

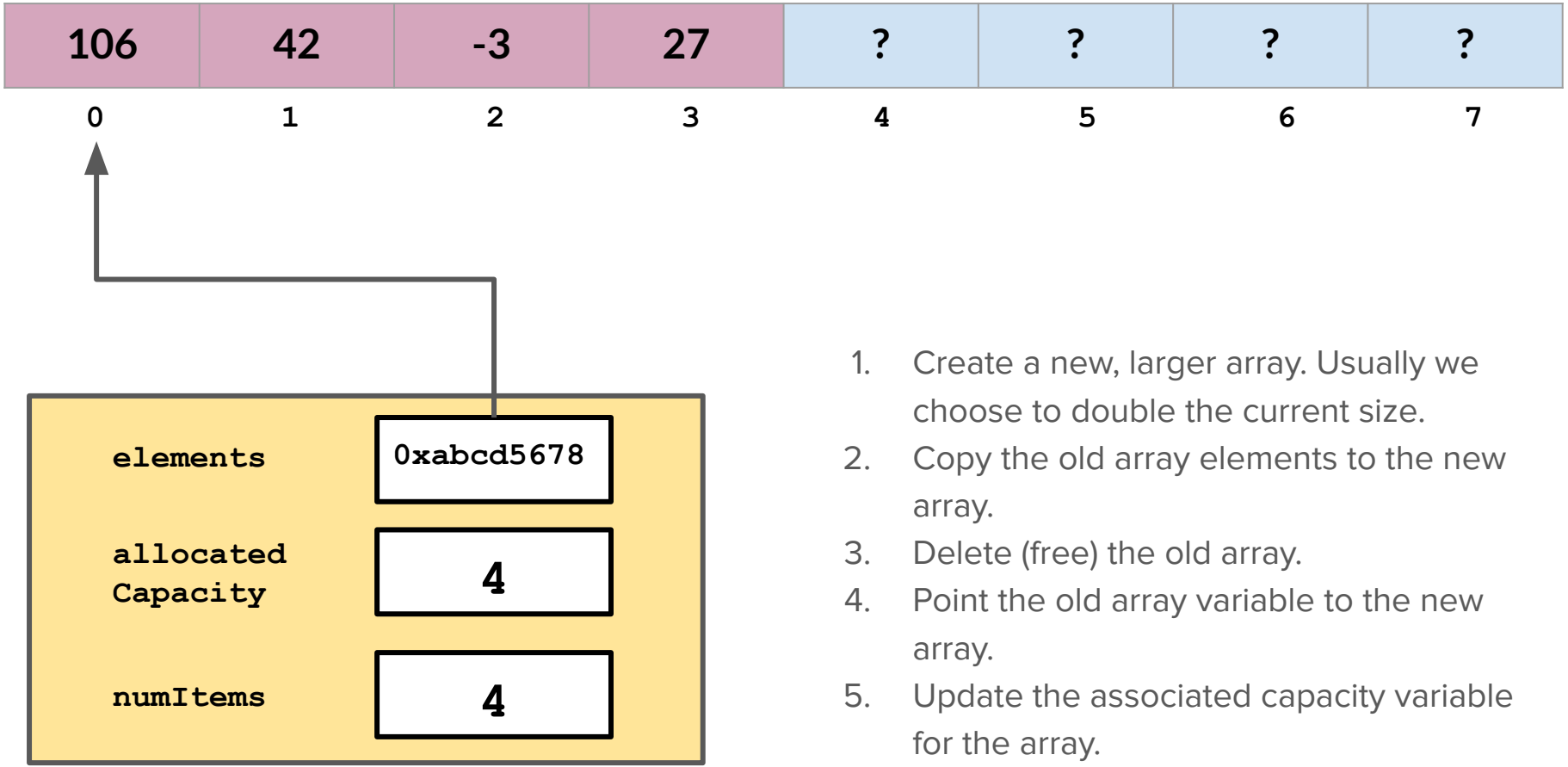
106	42	-3	27	?	?	?	?
0	1	2	3	4	5	6	7



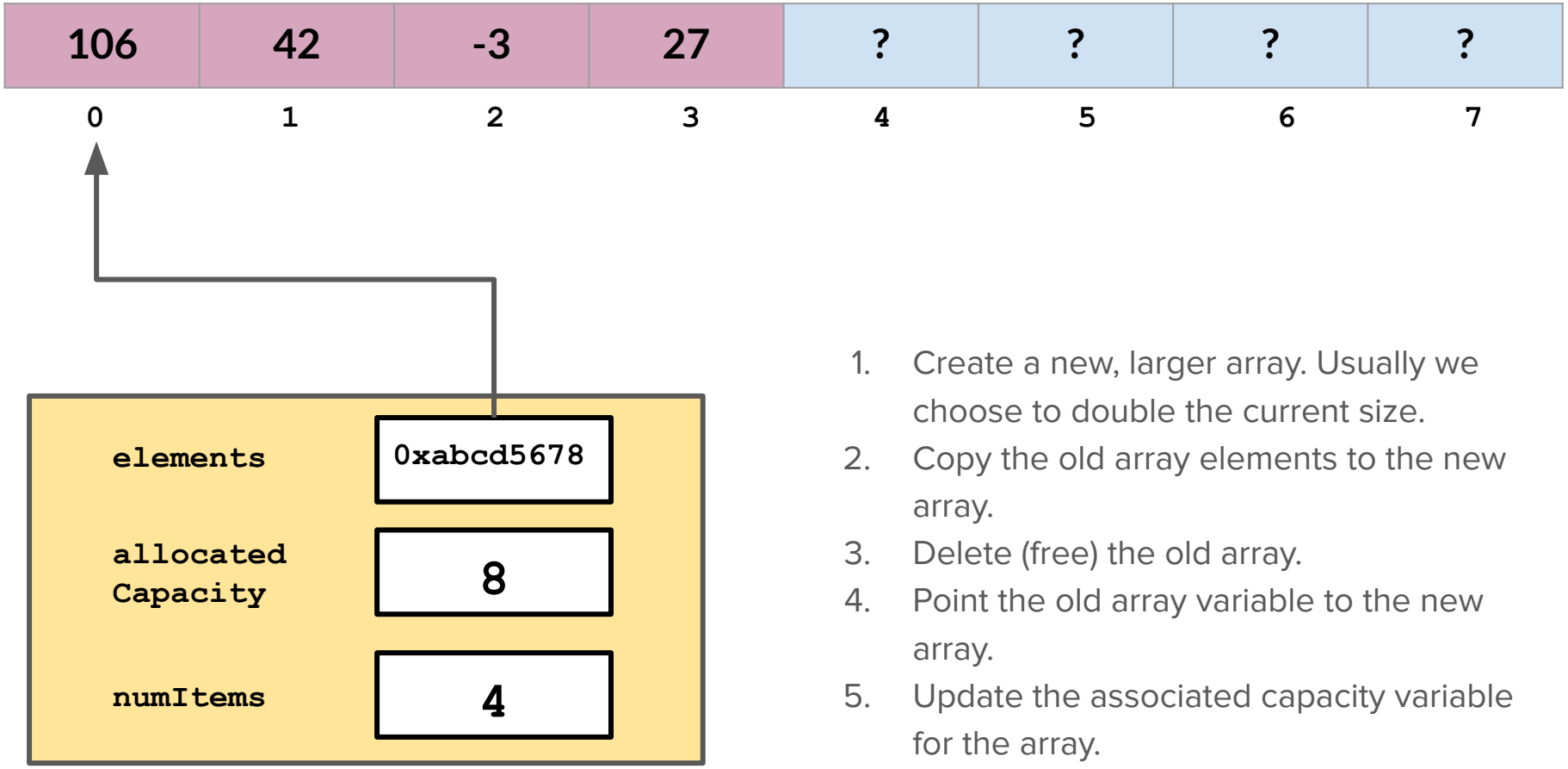
1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.
4. Point the old array variable to the new array.



1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.
4. Point the old array variable to the new array.



1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.
4. Point the old array variable to the new array.
5. Update the associated capacity variable for the array.



1. Create a new, larger array. Usually we choose to double the current size.
2. Copy the old array elements to the new array.
3. Delete (free) the old array.
4. Point the old array variable to the new array.
5. Update the associated capacity variable for the array.

Let's Code It! (Part 3)

`expand()` private helper function

Summary

Implementing ADT Classes

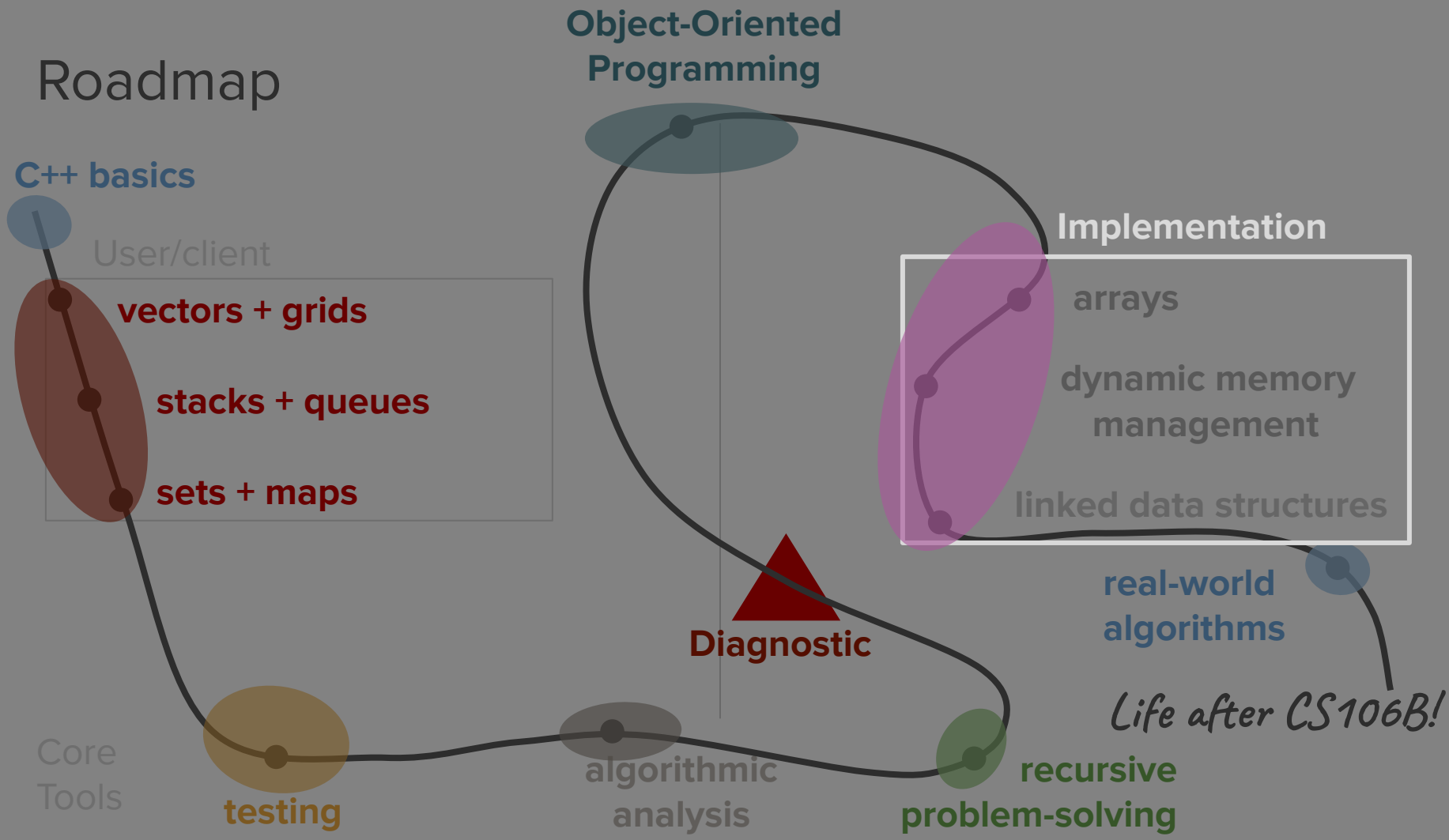
- The first step of implementing an ADT class (as with any class) is answering the three important questions regarding its public interface, private member variables, and initialization procedures.

- Most ADT classes will need to store their data in an underlying array. The organizational patterns of data in that array may vary, so it is important to illustrate and visualize the contents and any operations that may be done.

- The paradigm of "growable" arrays allows for fast and flexible containers with dynamic resizing capabilities that enable storage of large amounts of data.

What’s next?

Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Priority Queues and Heaps