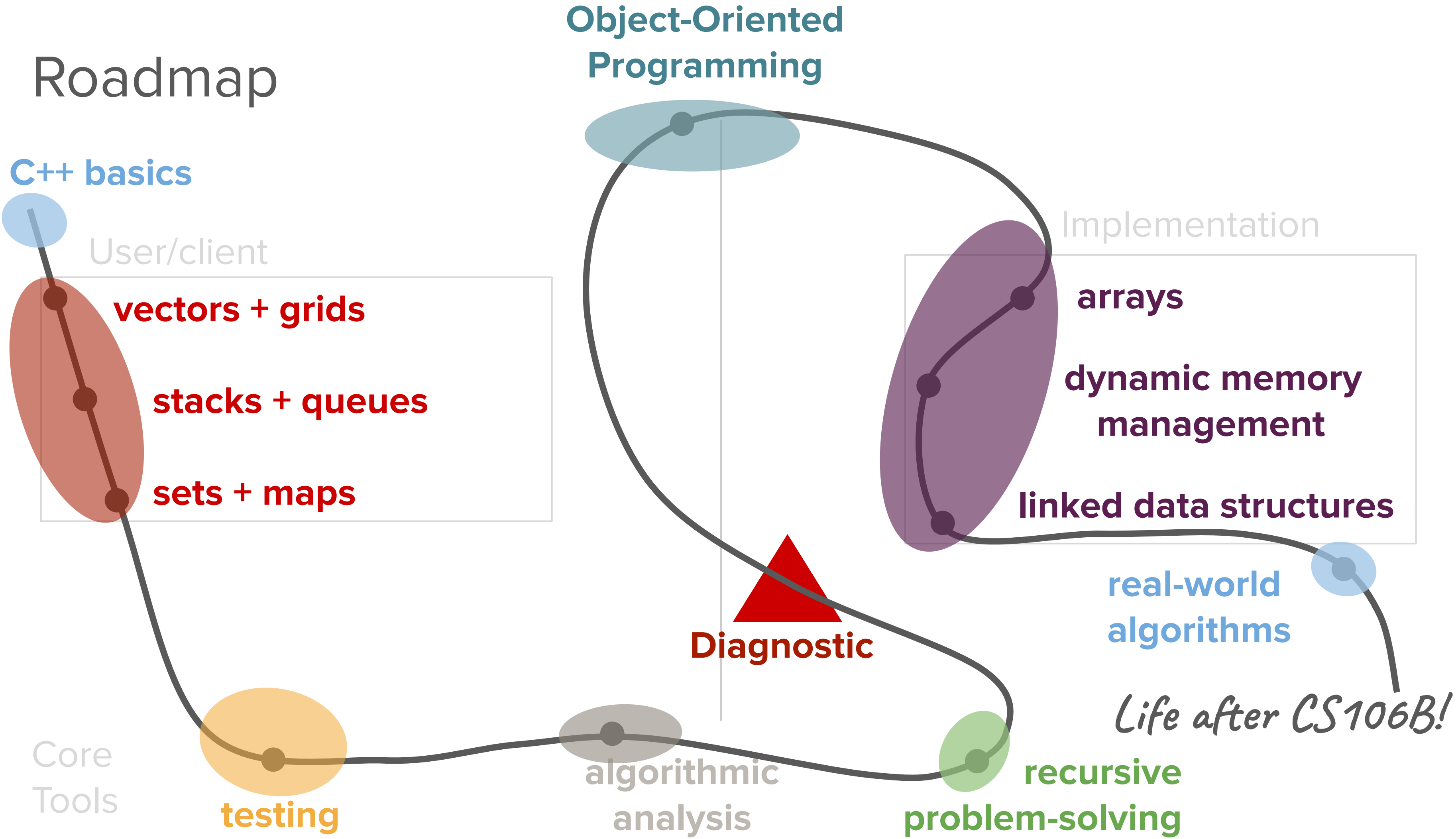
Dynamic Memory and Arrays

Having completing the diagnostic, what is one area of strength that you identified?

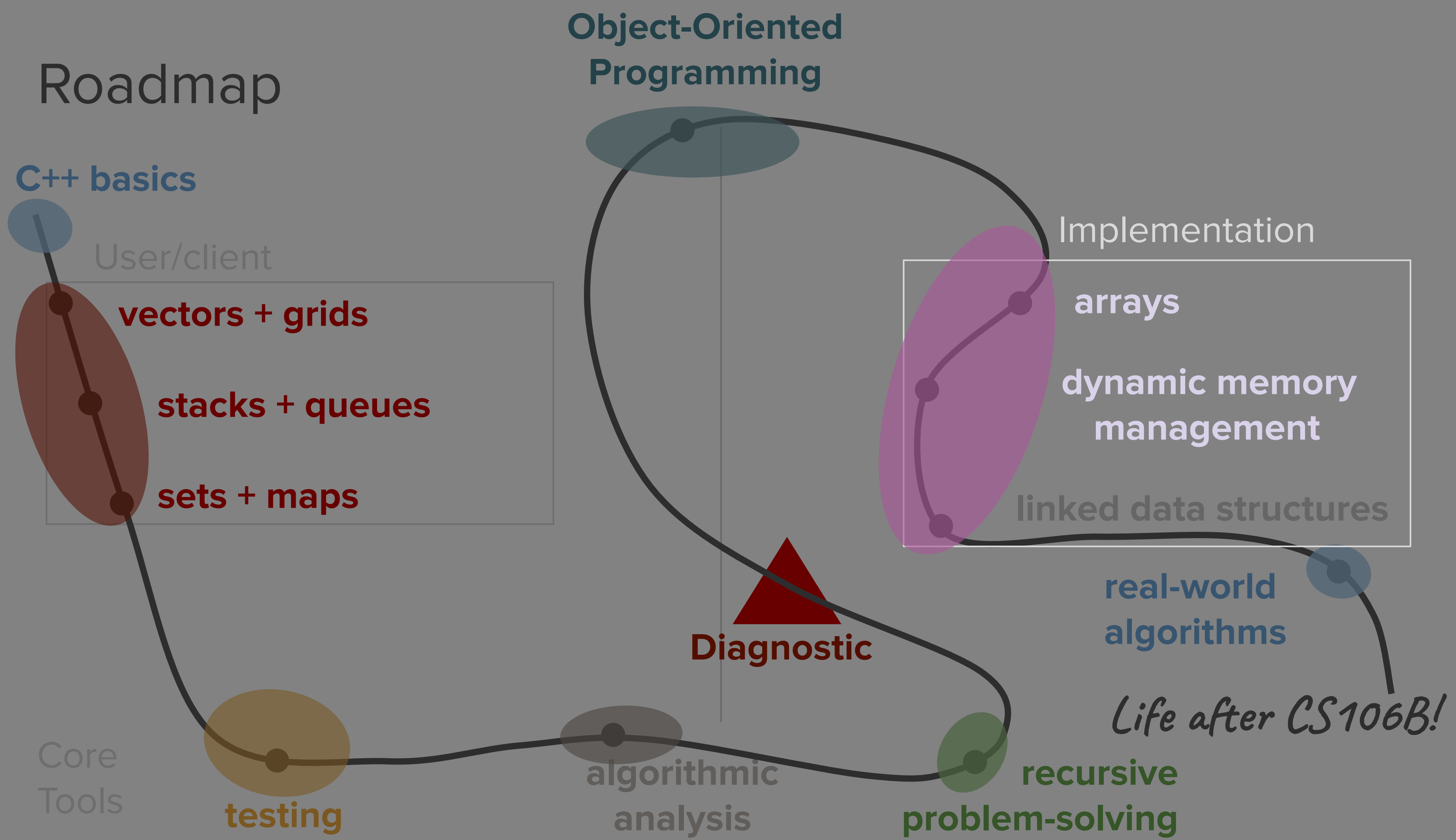
(no need to put your answer in the chat)

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

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

Today's question

What are the fundamental building blocks of data storage provided by C++?

Today's topics

1. Review
2. Classes Wrap-up (Bank Account)
3. Dynamic Allocation and Arrays
4. Implementing OurVector

Review

How do we accomplish this in C++? With classes!

Definition

abstraction

Design that hides the details of how something works while still allowing the user to access complex functionality

Definition

class

A class defines a new data type for our programs to use.

Definition

encapsulation

The process of grouping related information and relevant functions into one unit and defining where that information is accessible

What is a class?

- Examples of classes we've already seen: `Vector`s, `Map`s, `Stack`s, `Queue`s

- Every class has two parts:
    - an **interface** specifying what operations can be performed on instances of the class (this defines the abstraction boundary)
    - an **implementation** specifying how those operations are to be performed

- The only difference between structs + classes are the **encapsulation** defaults.
    - A struct defaults to **public** members (accessible outside the class itself).
    - A class defaults to **private** members (accessible only inside the class implementation).

Another way to think about classes...

- A blueprint for a new type of C++ **object**!
  - The blueprint describes a general structure, and we can create specific **instances** of our class using this structure.

Definition

instance

When we create an object that is our new type, we call this creating an instance of our class.

Three main parts

- Member variables
  - These are the variables stored within the class
  - Usually not accessible outside the class implementation

- Member functions (methods)
  - Functions you can call on the object
  - E.g. **`vec.add()`**, **`vec.size()`**, **`vec.remove()`**, etc.

- Constructor
  - Gets called when you create the object
  - E.g. **`Vector<int> vec;`**

How do we design a class (key questions)?

We must specify the 3 parts:

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

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

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

In general, classes are useful in helping us with complex programs where information can be grouped into objects.

Classes in C++

- Defining a class in C++ (typically) requires two steps:
  - Create a **header file** (typically suffixed with **`.h`**) describing what operations the class can perform and what internal state it needs.
  - Create an **implementation file** (typically suffixed with **`.cpp`**) that contains the implementation of the class.

- Clients of the class can then include (using the **`#include`** directive) the header file to use the class.

Takeaways

- As a best practice, member variables should be **private**, and you can create public member functions to allow users to edit them

- Member functions have an implicit parameter that allows them to know what object they're operating on

- When you don't have a constructor, there's a default 0 argument constructor that instantiates all private member variables
    - (We'll see an explicit constructor today!)

An example:
Structs vs. classes
(BankAccount)

Takeaways

- The constructor is a specially defined method for classes that initializes the state of new objects as they are created.
  - Often accepts parameters for the initial state of the fields.
  - Special naming convention defined as **`ClassName()`**
  - You can never directly call a constructor, but one will always be called when declaring a new instance of an object

- **`this`**
  - Refers to the current instance of an object that a method is being called on
  - Similar to the **`self`** keyword in Python and the **`this`** keyword in Java
  - Syntax: **`this->memberVariable`**
  - Common usage: In the constructor, so parameter names can match the names of the object's member variables.

Announcements

Announcements

- Congratulations on finishing the diagnostic! The course staff will be hard at work grading this week – we're aiming to have grades back by **the end of the week.**

- Assignment 4 will be released at the end of the day**.**
    - The Assignment 4 YEAH session will be **Wednesday at TBD**. Look out for an announcement on Ed!

Where are we now?

classes

object-oriented programming

abstract data structures
(vectors, maps, etc.)

arrays

dynamic memory
management

linked data structures

testing

algorithmic analysis

recursive problem-solving

classes

object-oriented programming

abstract data structures
(vectors, maps, etc.)

arrays

dynamic memory
management

linked data structures

testing

algorithmic analysis

recursive problem-solving

classes

object-oriented programming

abstract data structures
(vectors, maps, etc.)

arrays

dynamic memory management

linked data structures

testing

algorithmic analysis

recursive problem-solving

classes

object-oriented programming

We've now crossed the abstraction boundary!

abstract data structures
(vectors, maps, etc.)

arrays

dynamic memory management

linked data structures

testing

algorithmic analysis

recursive problem-solving

RandomBag Revisited

```cpp
#pragma once
#include "vector.h"

class RandomBag {
public:
    void add(int value);
    int removeRandom();
    int size() const;
    bool isEmpty() const;

private:
    Vector<int> elems;
};
```

```
#pragma once
#include "vector.h"

class RandomBag {
public:
  void add(int value);
  int removeRandom();
  int size() const;
  bool isEmpty() const;

private:
  Vector<int> elems;
};
```

Turtles All the Way Down?

- Last time, we implemented a **`RandomBag`** on top of our library **`Vector`** type.

Turtles All the Way Down?

- Last time, we implemented a **`RandomBag`** on top of our library **`Vector`** type.

- But the **`Vector`** type is itself an abstraction (provided library) – what is it layered on top of?

Turtles All the Way Down?

- Last time, we implemented a **`RandomBag`** on top of our library **`Vector`** type.
- But the **`Vector`** type is itself an abstraction (provided library) – what is it layered on top of?
- **Question:** What are the fundamental building blocks provided by the language, and how do we use them to build our own custom classes?

What are the fundamental building blocks of data storage provided by C++?

Getting Storage Space

- The **`Vector`**, **`Stack`**, **`Queue`**, etc. all need storage space to put the elements that they store.

Getting Storage Space

- The **`Vector`**, **`Stack`**, **`Queue`**, etc. all need storage space to put the elements that they store.

- That storage space is acquired using **dynamic memory allocation**.

Getting Storage Space

- The **`Vector`**, **`Stack`**, **`Queue`**, etc. all need storage space to put the elements that they store.

- That storage space is acquired using **dynamic memory allocation**.

- Essentially:
  - 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.

Arrays

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

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.

Arrays

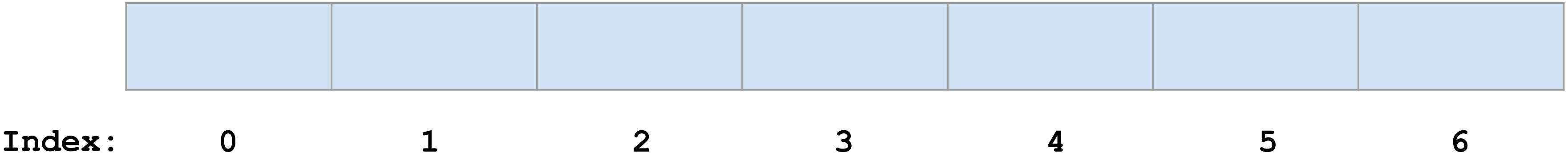
- Storage space on computers, which we often refer to as memory, is allocated in organized chunks called **arrays**

- 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.

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.

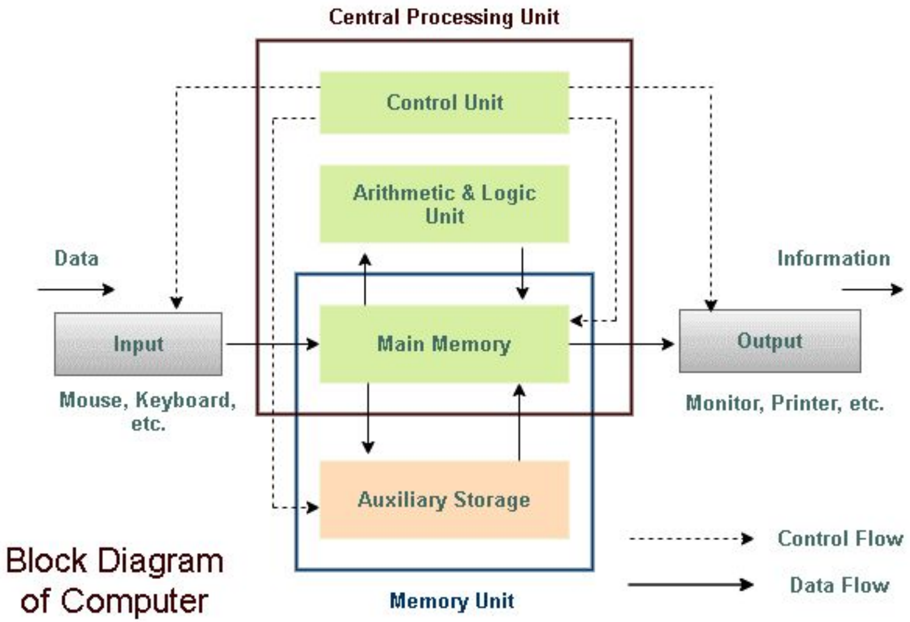


Dynamically Allocating Arrays

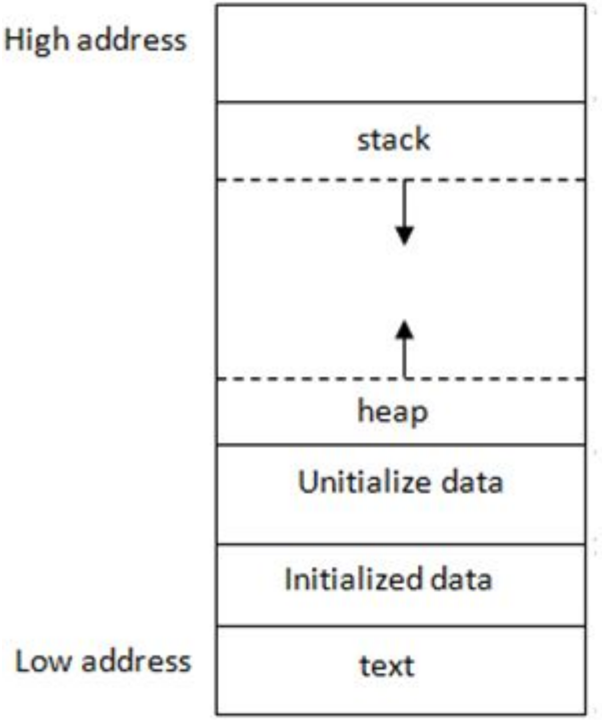
- We are going to be **explicitly** asking the runtime for a new piece of storage space.
- This isn't like the memory you've used before, so it actually lives in a **new region of your computer's memory.**

Pit stop: what does computer memory actually look like?

Computer Organization



Computer Organization (Main Memory)



- All of the variables we have used so far have been stored in the contiguous “array” coming from the **Stack** region.
- When we request **dynamic memory**, we request it from the **Heap** region.
    - We separate the **Stack** and the **Heap** because dynamic (**Heap**) memory behaves a little differently than **Stack** memory.

Dynamically Allocating Arrays

- To request a dynamic (**Heap**) memory array, you need to use the **`new`** keyword. You also want to specify a type, along with the number of memory spaces you want to use.
  - ex. **`= new int[5];`**
- Now, here’s the weird part:

Dynamically Allocating Arrays

- To request a dynamic (**Heap**) memory array, you need to use the **`new`** keyword. You also want to specify a type, along with the number of memory spaces you want to use.
  - ex. **`= new int[5];`**
- Because the array lives on the **Heap** (which is far away!), you can't access it like a normal variable. You need a reference, or a **Pointer** to the array. If the array elements have type **`T`**, the pointer will have type **`T*`**.
  - e.g. **`int*, string*, Vector<double>*`**

Dynamically Allocating Arrays

- Here's how you might request a dynamic (or Heap-allocated) array:
- 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.

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.

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 is essentially **your data’s index in the computer’s main memory array.**

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 essentially the **your data’s index in the computer’s main memory array.**

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

Dynamic Allocation Demo

```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

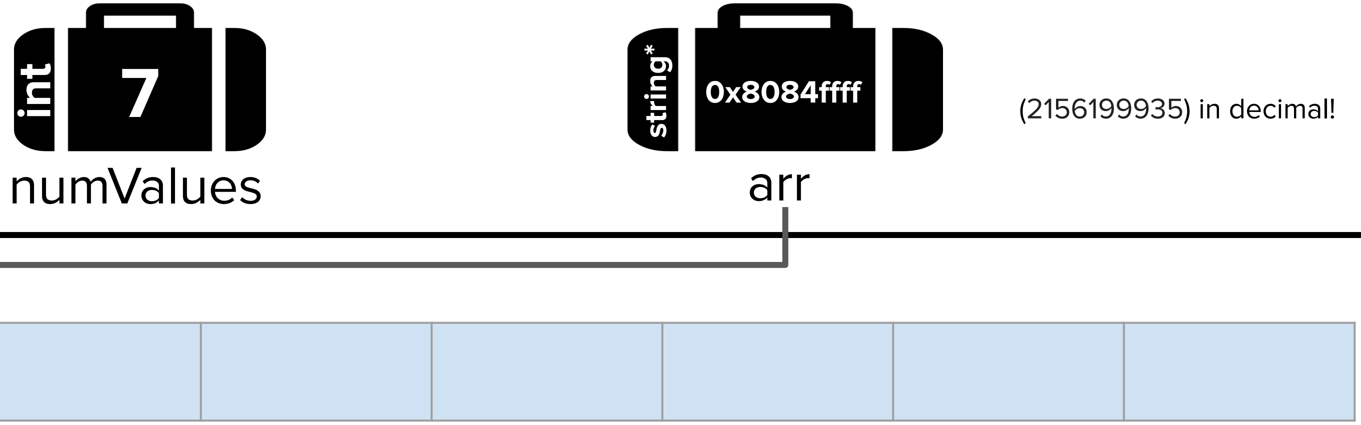
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

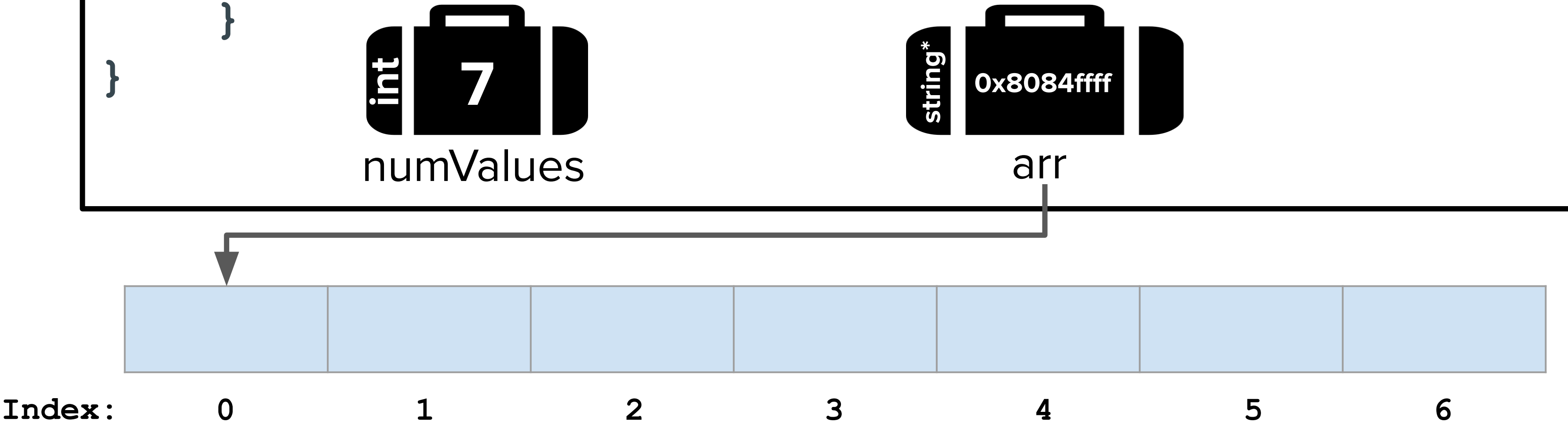
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

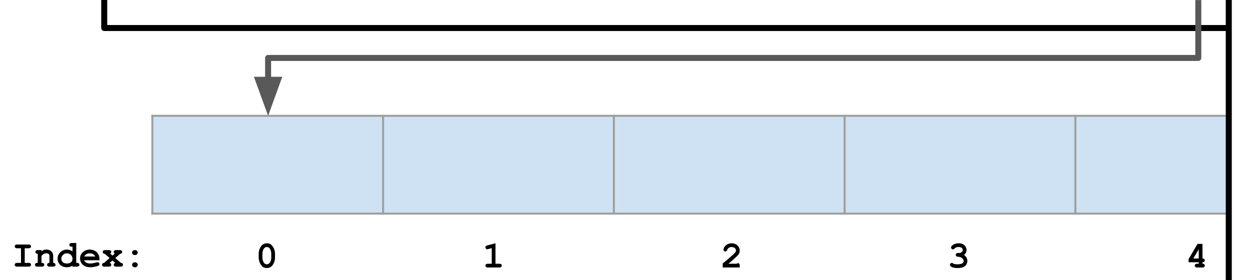
```cpp
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

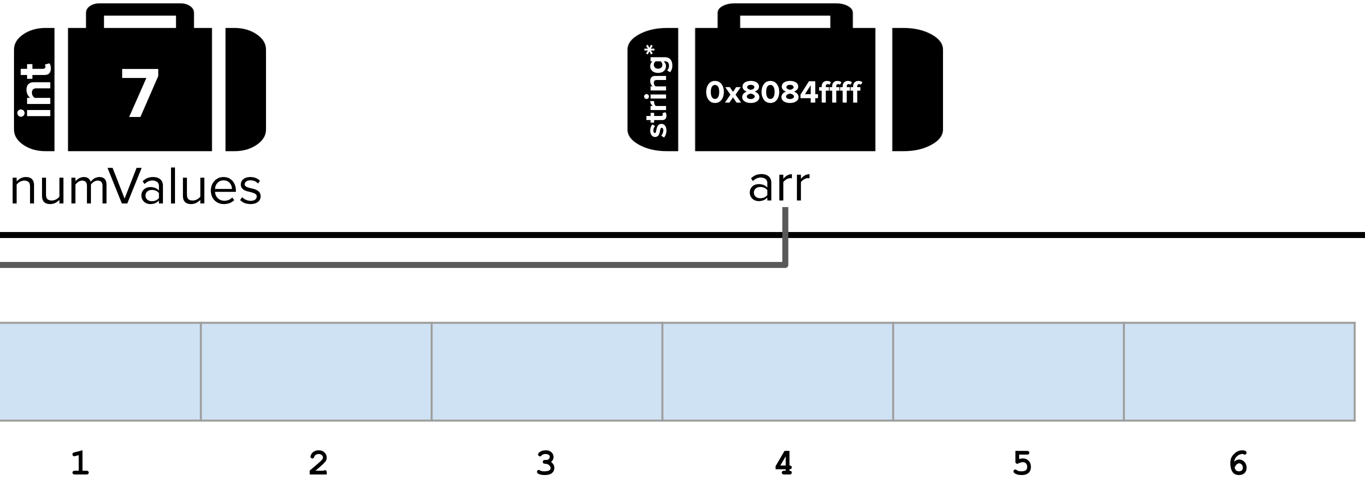


```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

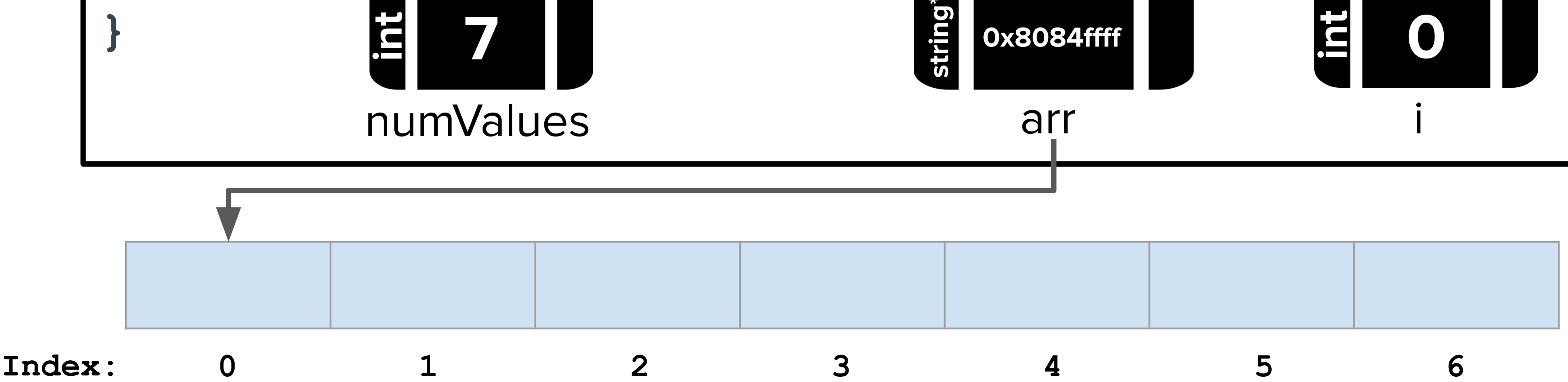


Because the variable arr points to the array, it is called a ***pointer.***

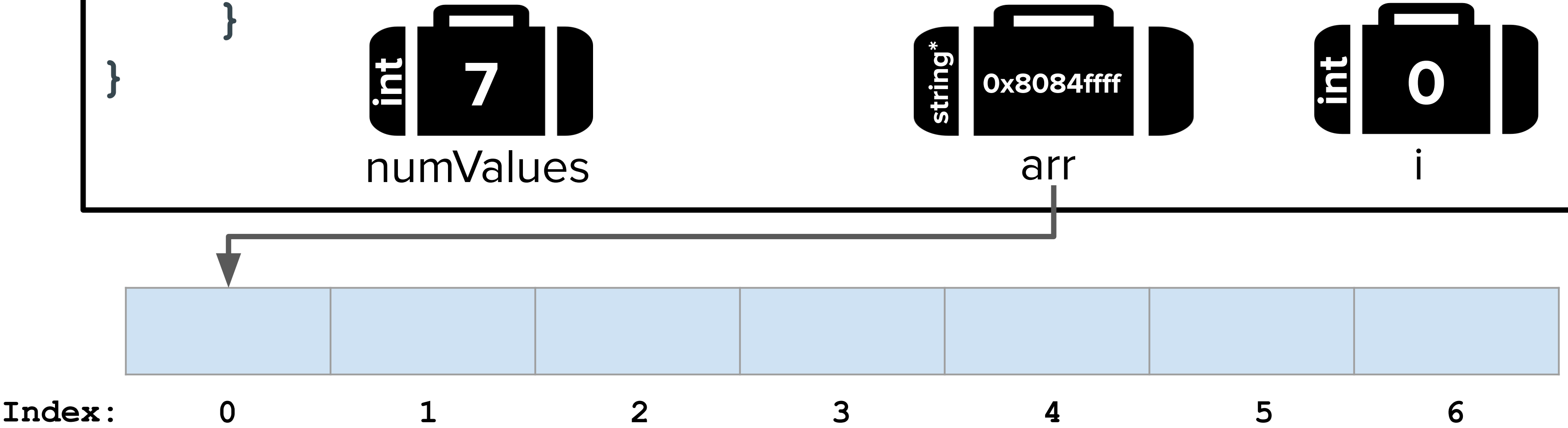
```cpp
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



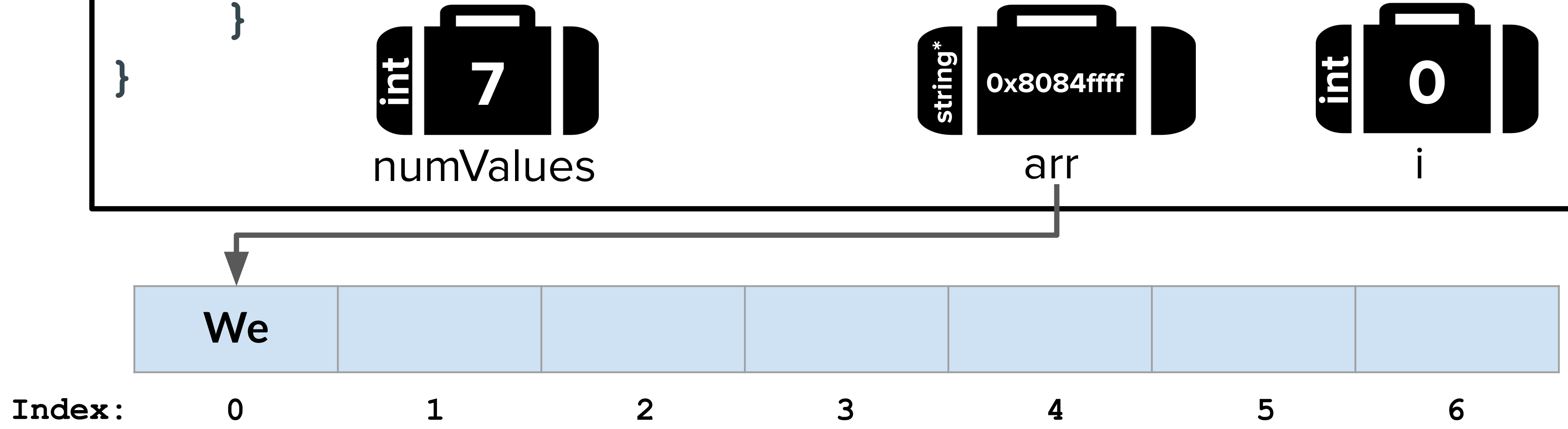
```cpp
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



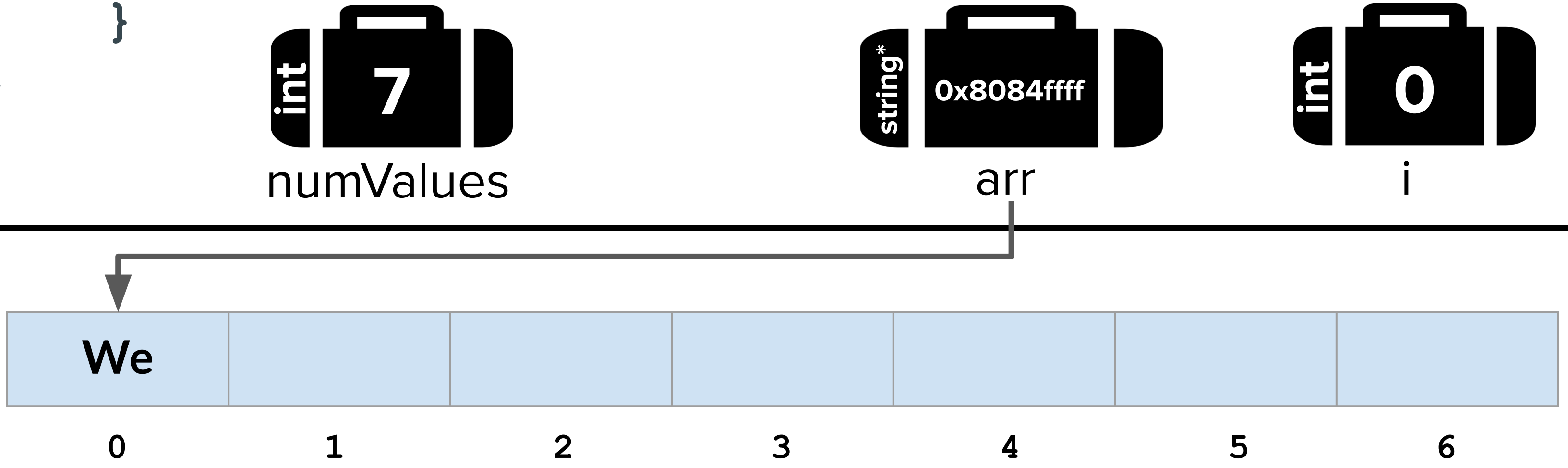
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



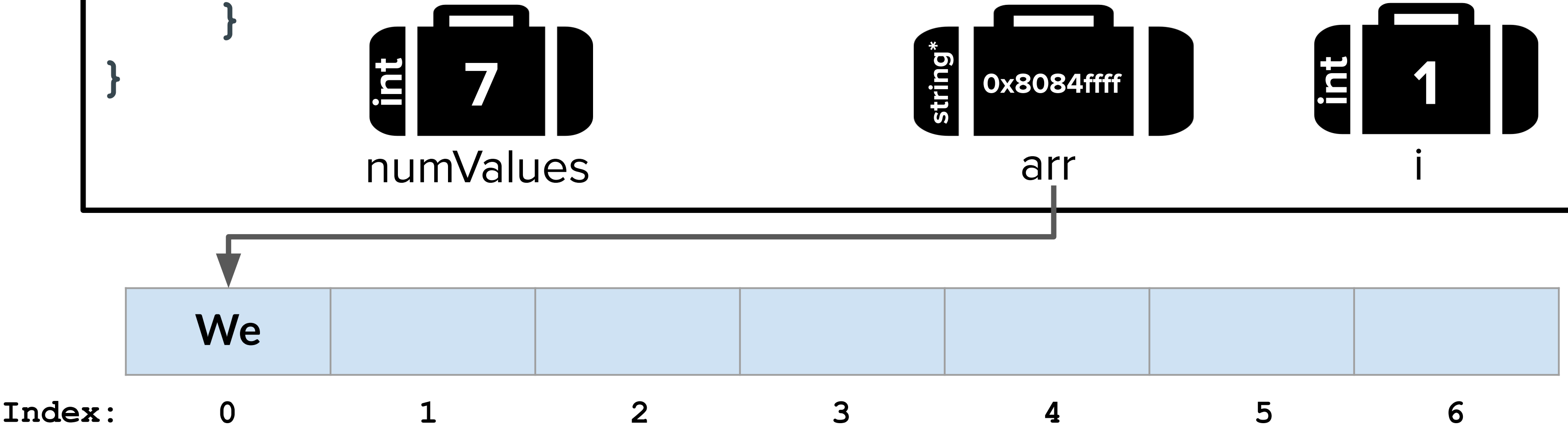
```cpp
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



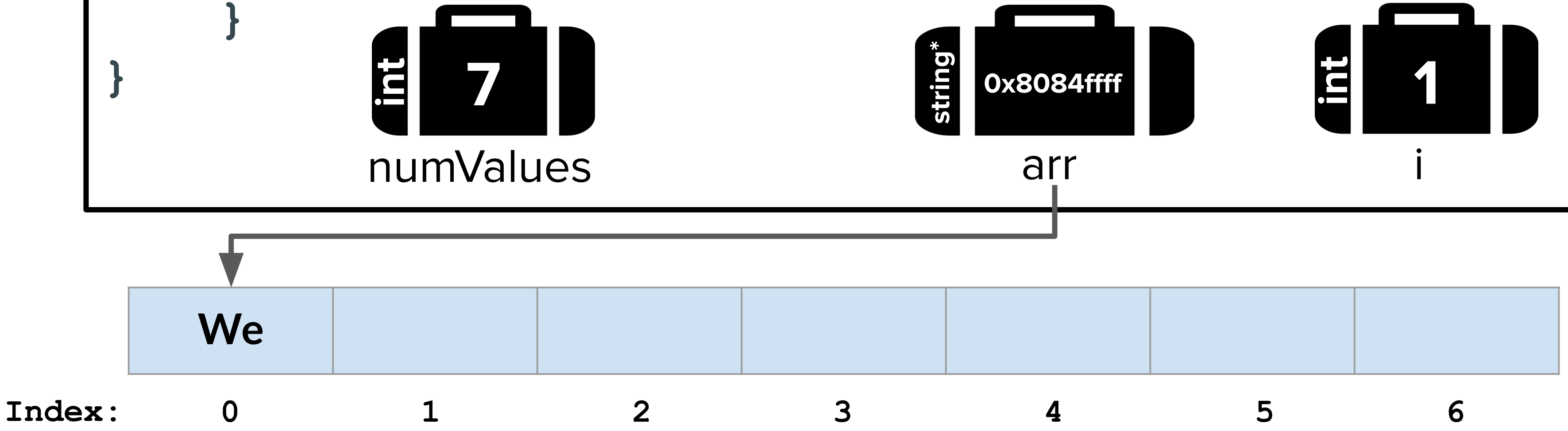
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



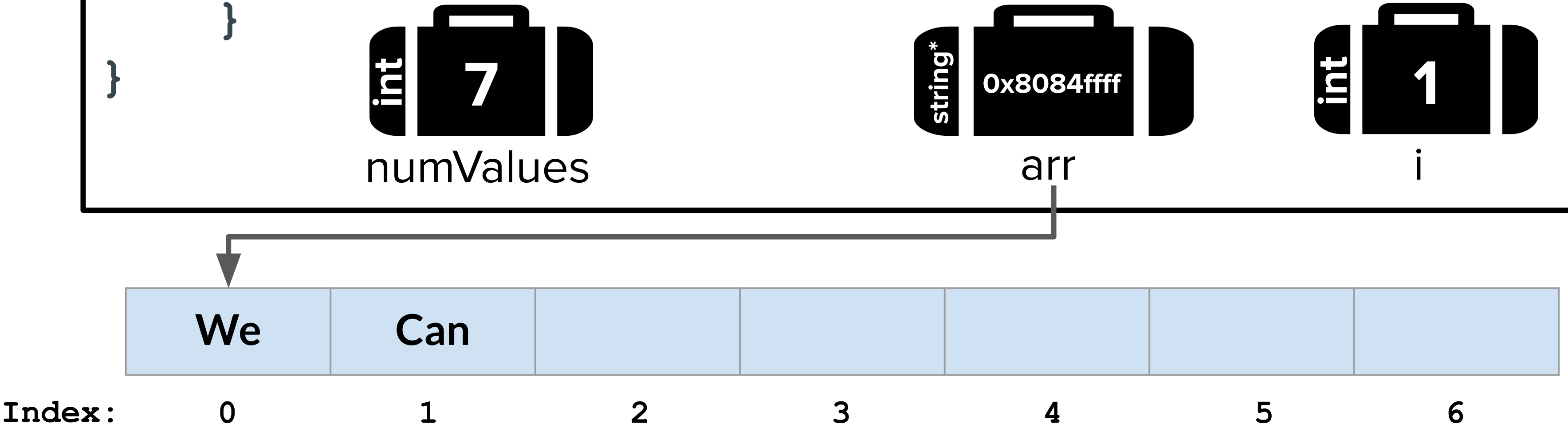
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



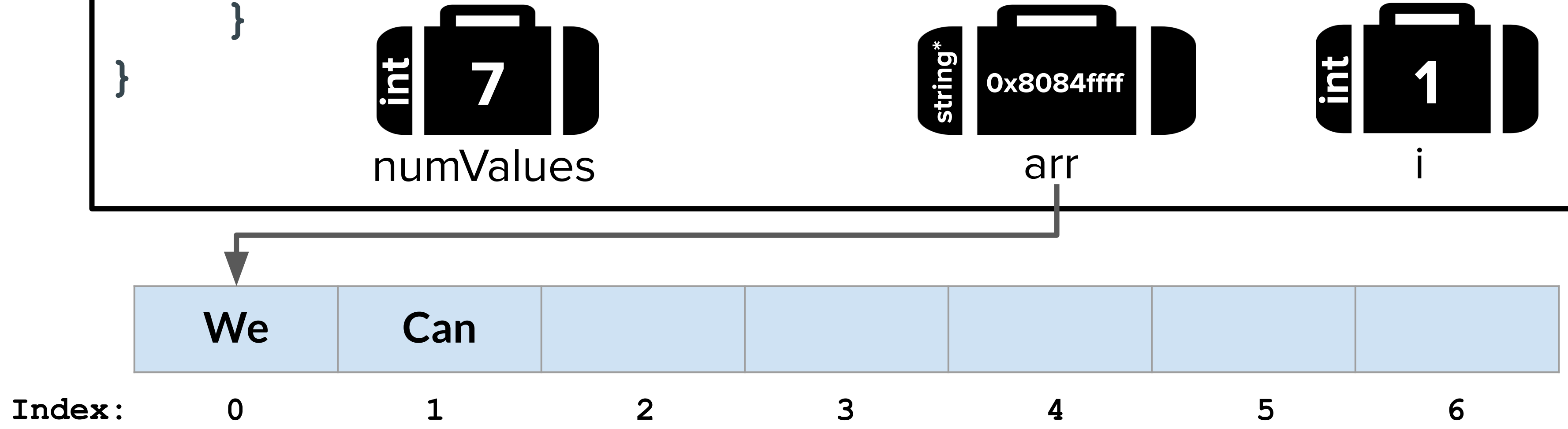
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



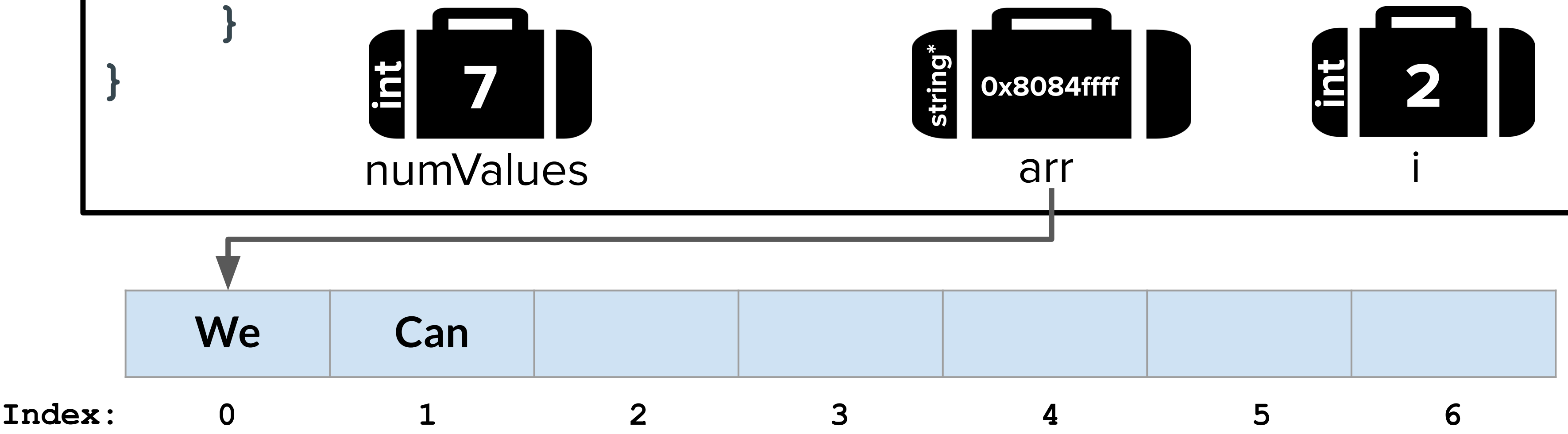
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



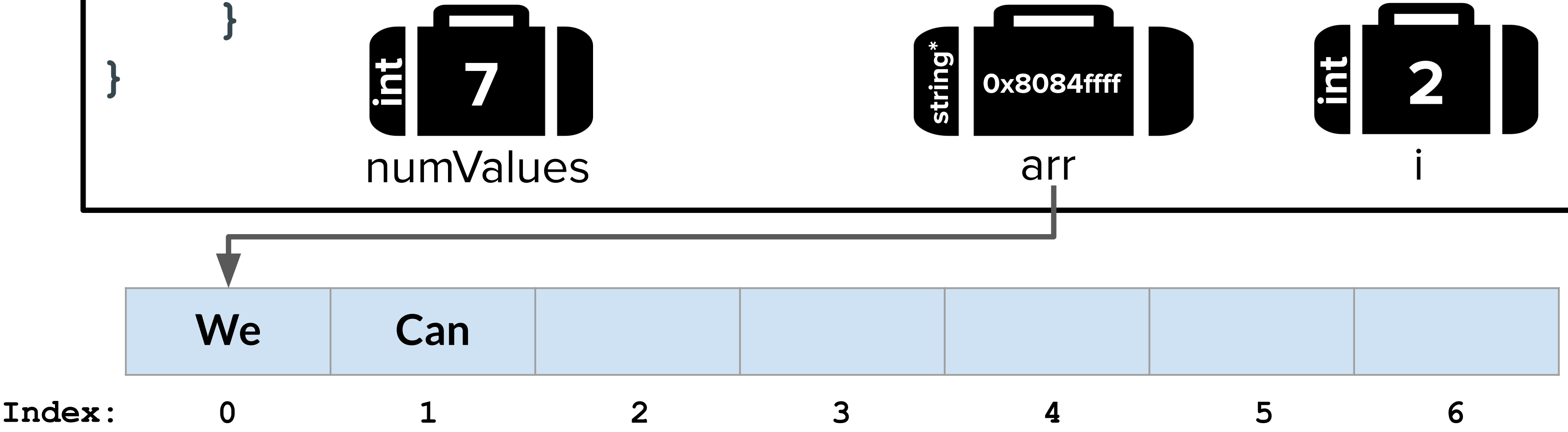
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



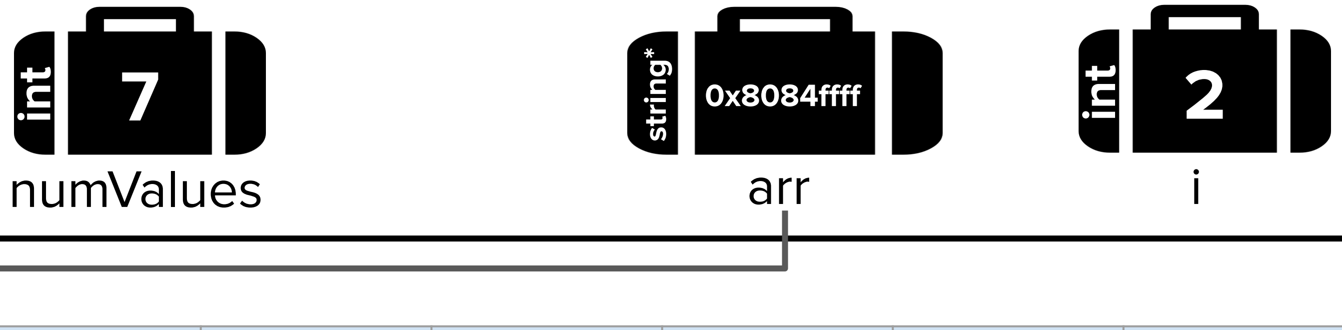
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

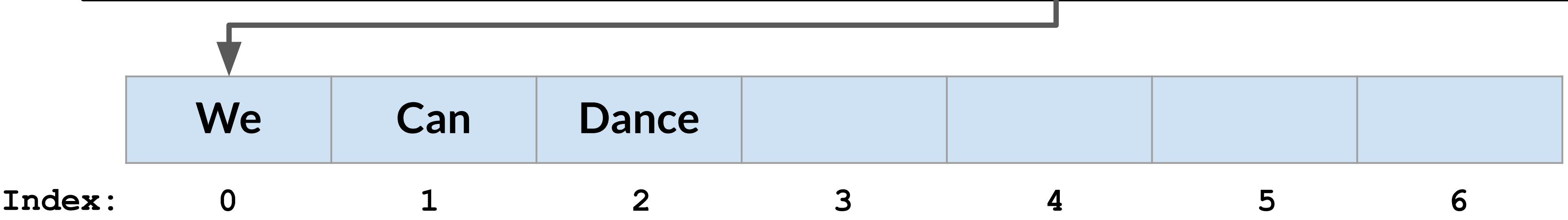


```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

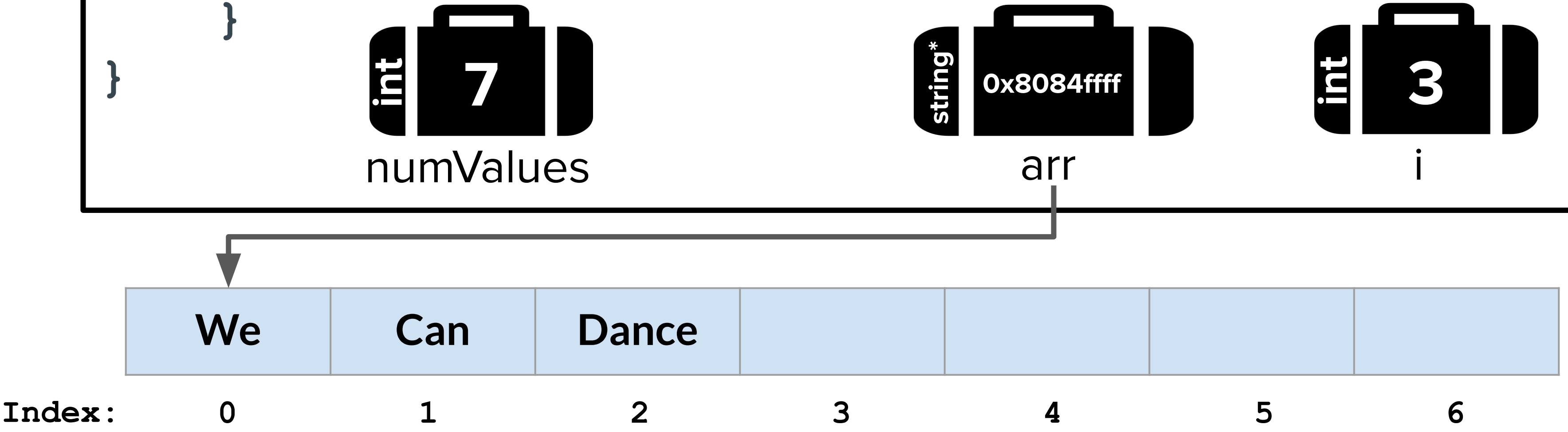


	We	Can	Dance				
Index:	0	1	2	3	4	5	6

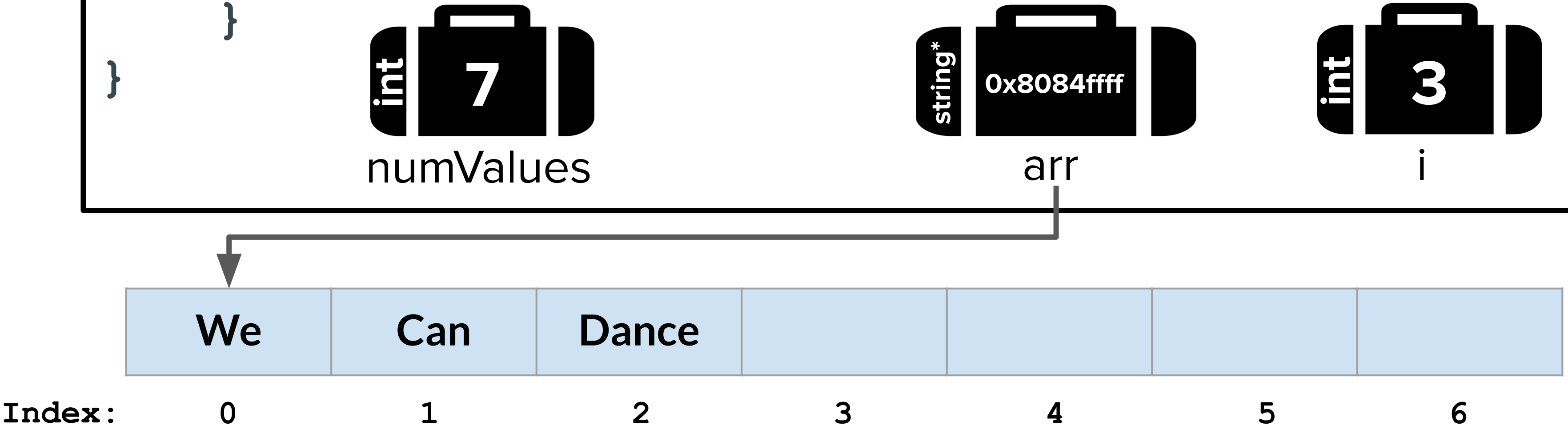
```cpp
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



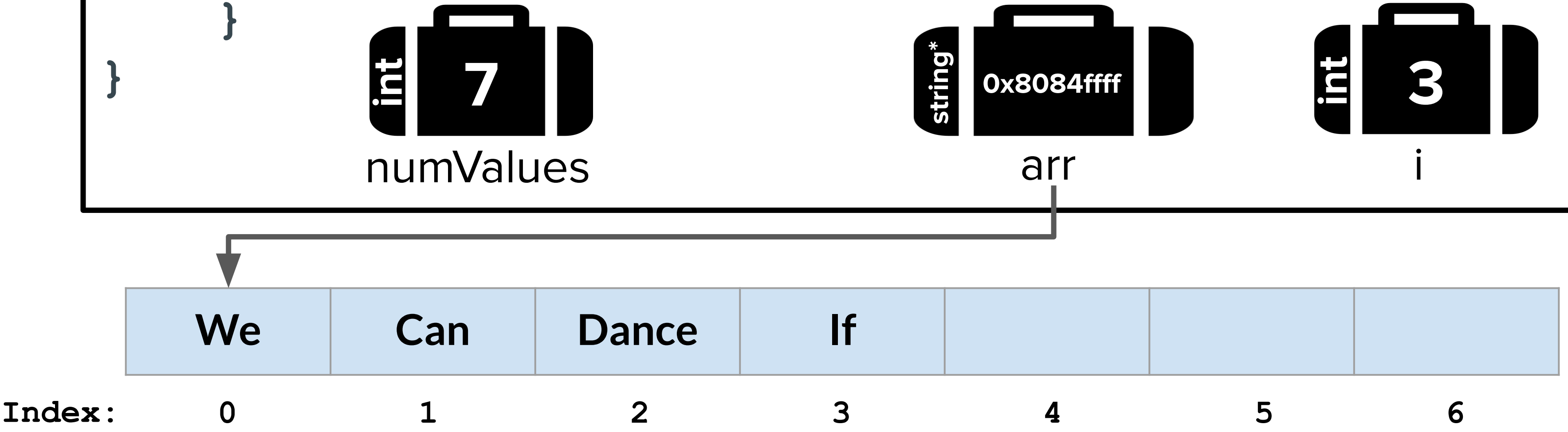
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



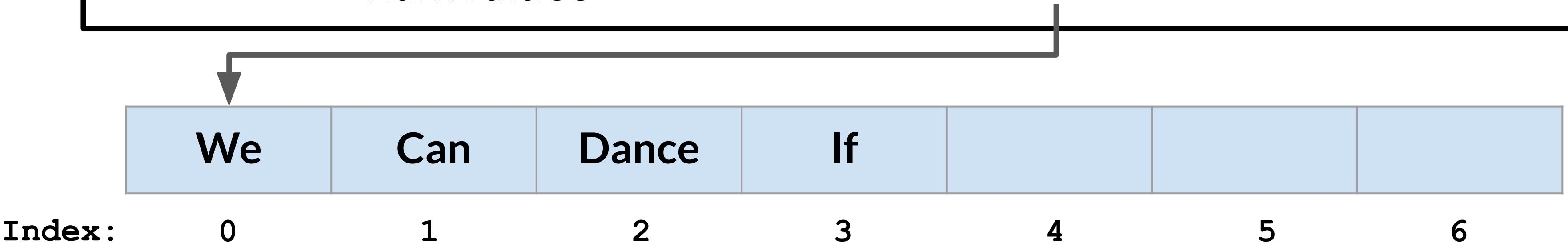
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



```cpp
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



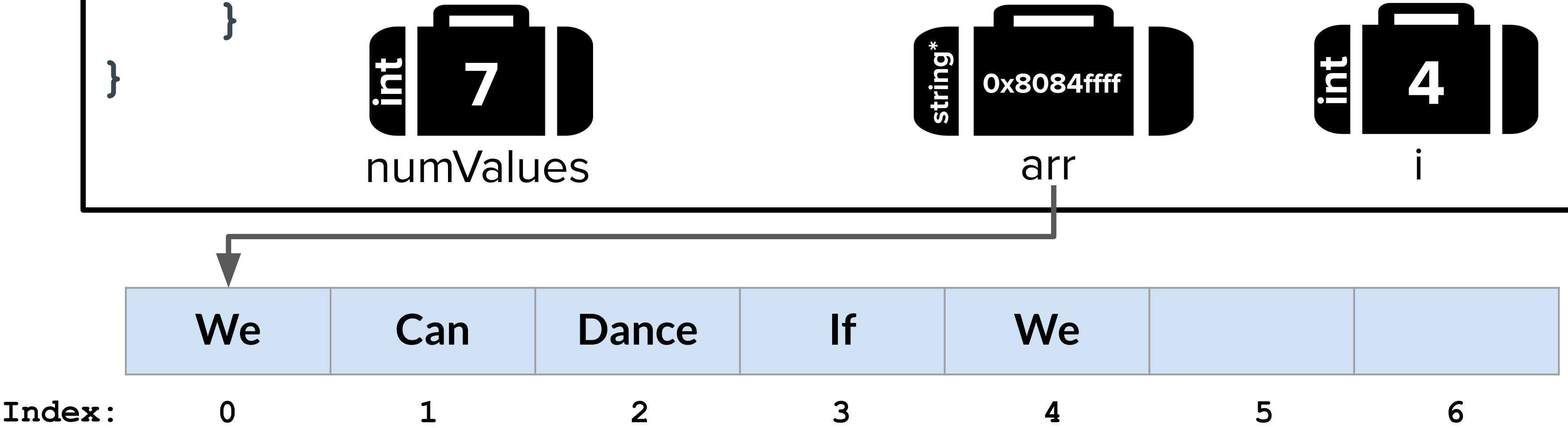
```cpp
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

Index:	0	1	2	3	4	5	6
	We	Can	Dance	If			

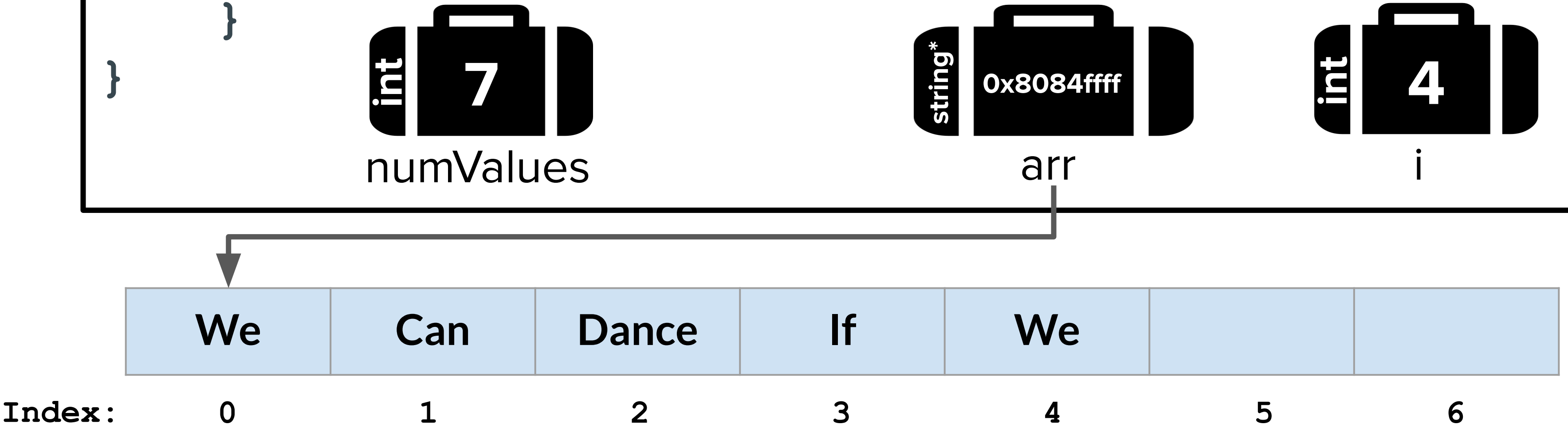
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

	We	Can	Dance	If			
Index:	0	1	2	3	4	5	6

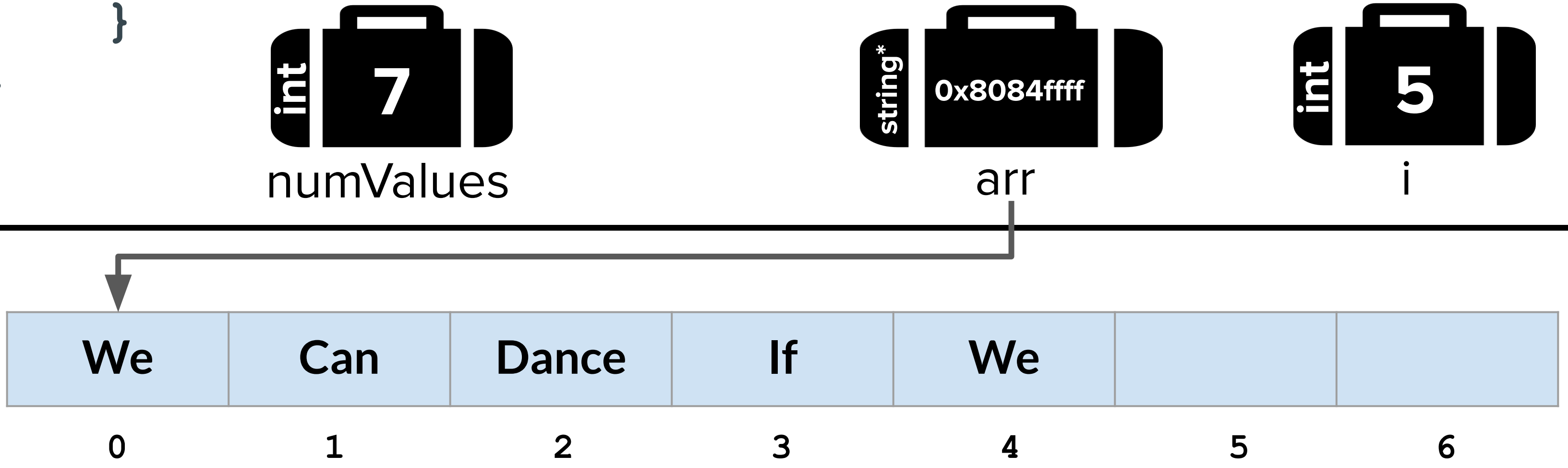
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



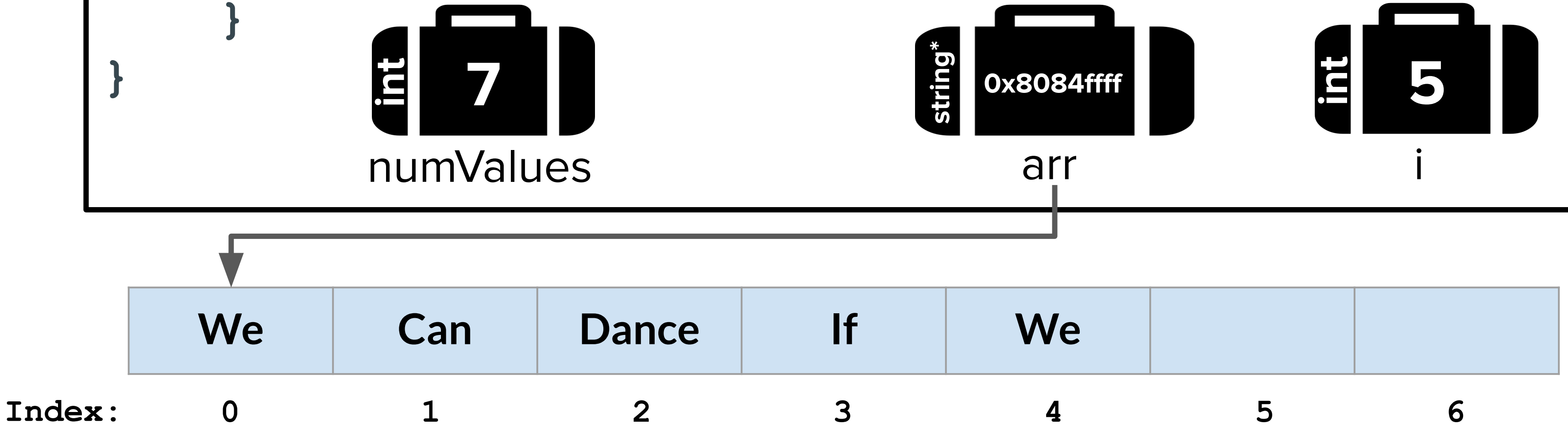
```cpp
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



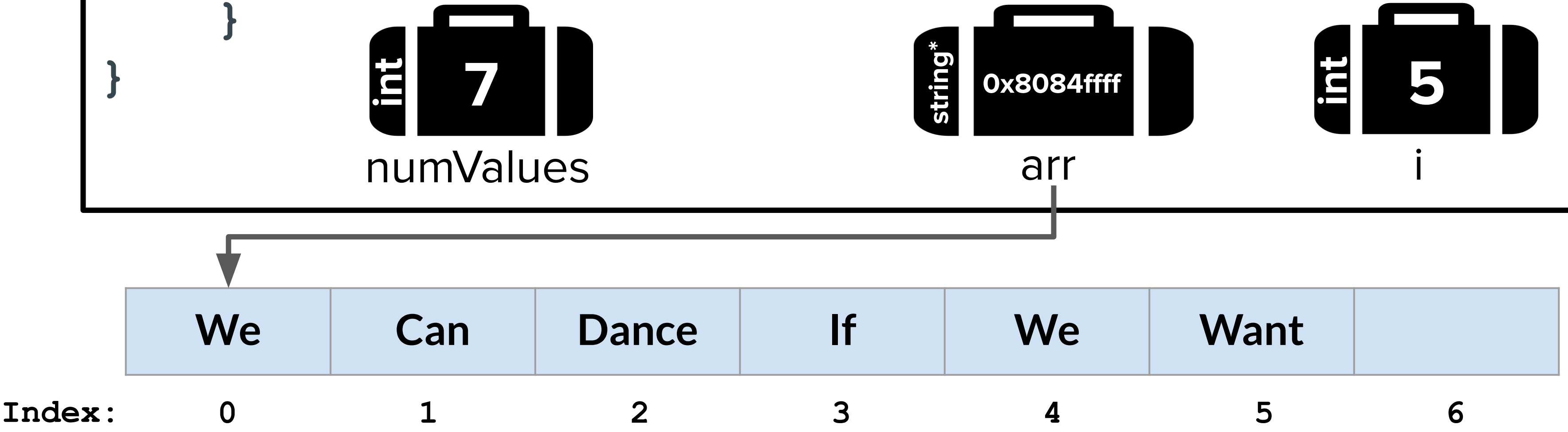
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



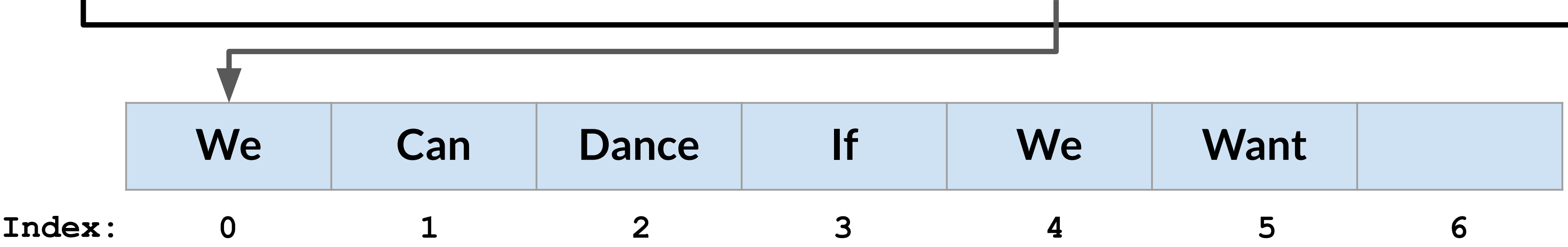
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

	We	Can	Dance	If	We	Want	
Index:	0	1	2	3	4	5	6

```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

	We	Can	Dance	If	We	Want	
Index:	0	1	2	3	4	5	6

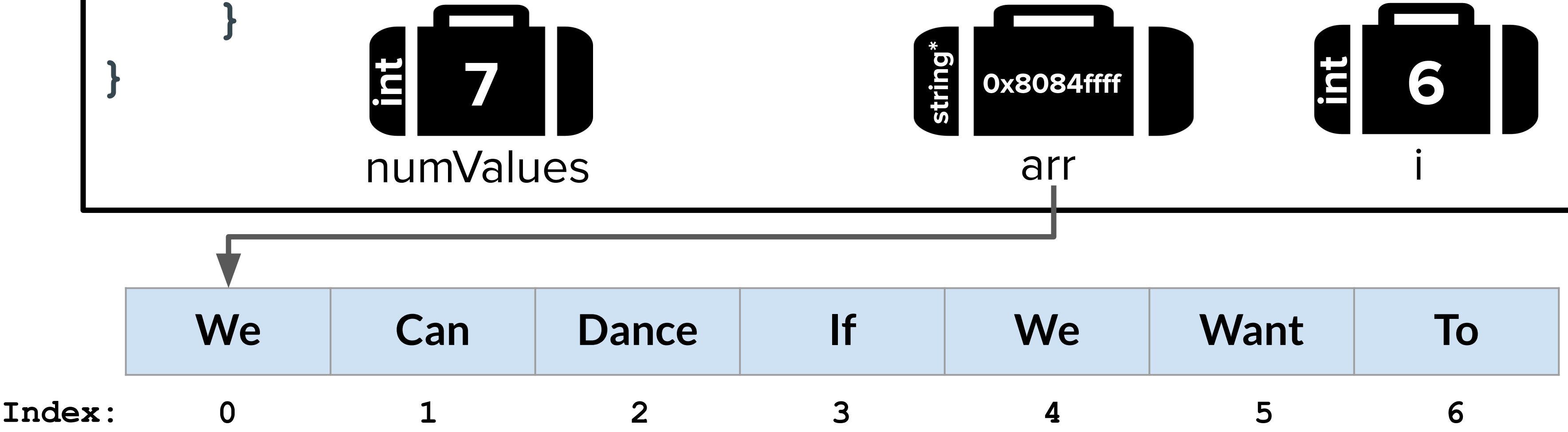
```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



```cpp
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

	We	Can	Dance	If	We	Want	To
Index:	0	1	2	3	4	5	6

```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```



```cpp
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

	We	Can	Dance	If	We	Want	To
Index:	0	1	2	3	4	5	6

```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

	We	Can	Dance	If	We	Want	To
Index:	0	1	2	3	4	5	6

```cpp
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++)
        arr[i] = getLine("Enter a string
    }
    for (int i = 0; i < numValues; i++)
        cout << i << ": " << arr[i] << e
    }
}
```

```
0: We
1: Can
2: Dance
3: If
4: We
5: Want
6: To
```

	We	Can	Dance	If	We	Want	To
Index:	0	1	2	3	4	5	6

```
int main() {
    int numValues = getInteger("How many lines? ");
    string* arr = new string[numValues];
    for (int i = 0; i < numValues; i++) {
        arr[i] = getLine("Enter a string: ");
    }
    for (int i = 0; i < numValues; i++) {
        cout << i << ": " << arr[i] << endl;
    }
}
```

	We	Can	Dance	If	We	Want	To
Index:	0	1	2	3	4	5	6

Dynamically Allocating Arrays

- C++'s language philosophy prioritizes speed over safety and simplicity.

Dynamically Allocating Arrays

- C++'s language philosophy prioritizes speed over safety and simplicity.

- 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."
    - Question for you: how do things like Vectors and Stacks have "infinite" size?

Dynamically Allocating Arrays

- C++'s language philosophy prioritizes speed over safety and simplicity.

- 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."
  - Question for you: how do things like Vectors and Stacks have "infinite" size?
- 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, or you make the front page of the New York Times...

"All the News That's Fit to Print"

The New York Times

Late Edition

New York: Today, partly sunny, milder. High 59-64. Tonight, mostly cloudy. Low 48-54. Tomorrow, cloudy, windy, rain developing. High 57-62. Yesterday: High 58, low 41. Details, page D16.

VOL. CXXXVIII . . . No. 47,679 — NEW YORK, FRIDAY, NOVEMBER 4, 1988 — 35 CENTS

Gov. Michael S. Dukakis having his picture taken by a 10-year-old fan at a town meeting in Fairless Hills, Pa., during a tour of the Northeast in which he emphasized the drug problem. Page A19. Vice President Bush addressed supporters a rally in Columbus, Ohio. Less than a week after Mr. Dukakis acknowledged being a liberal, Mr. Bush said yesterday that "this election is not about labels." Page A18.

'Virus' in Military Computers Disrupts Systems Nationwide

By JOHN MARKOFF

In an intrusion that raises questions about the vulnerability of the nation's computers, a Department of Defense network has been disrupted since Wednesday by a rapidly spreading "virus" program apparently introduced by a computer science student.

The program reproduced itself through the computer network, making hundreds of copies in each machine it reached, effectively clogging systems linking thousands of military, corporate and university computers around the nation and preventing them from doing additional work. The virus is thought not to have destroyed any files.

By late yesterday afternoon computer experts were calling the virus the largest assault ever on the nation's computers.

'The Big Issue'

"The big issue is that a relatively benign software program can virtually bring our computing community to its knees and keep it there for some time," said Chuck Cole, deputy computer security manager at Lawrence Livermore Laboratory in Livermore, Calif., one of the sites affected by the intrusion. "The cost is going to be staggering."

Clifford Stoll, a computer security expert at Harvard University, added: "There is not one system manager who is not tearing his hair out. It's causing enormous headaches."

The affected computers carry a tremendous variety of business and research information among military officials, researchers and corporations.

While some sensitive military data are involved, the computers handling the nation's most sensitive secret information, like that on the control of nuclear weapons, are thought not to have been touched by the virus.

Parallel to Biological Virus

Computer viruses are so named because they parallel in the computer world the behavior of biological viruses. A virus is a program, or a set of instructions to a computer, that is either planted on a floppy disk meant to be used with the computer or introduced when the computer is communicating over telephone lines or data networks with other computers.

The programs can copy themselves into the computer's master software, or operating system, usually without calling any attention to themselves. From there, the program can be passed to additional computers.

Depending upon the intent of the software's creator, the program might cause a provocative but otherwise harmless message to appear on the computer's screen. Or it could systematically destroy data in the computer's memory. In this case, the virus program did nothing more than reproduce itself rapidly.

The program was apparently a result of an experiment, which

Continued on Page A21, Column 2

PENTAGON REPORTS IMPROPER CHARGES FOR CONSULTANTS

CONTRACTORS CRITICIZED

Inquiry Shows Routine Billing of Government by Industry on Fees, Some Dubious

By JOHN H. CUSHMAN Jr.

Special to The New York Times

WASHINGTON, Nov. 3 — A Pentagon investigation has found that the nation's largest military contractors routinely charge the Defense Department for hundreds of millions of dollars paid to consultants, often without justification.

The report of the investigation said that neither the military's current rules nor the contractors' own policies are adequate to assure that the Government does not improperly pay for privately arranged consulting work. Senior Defense Department officials said the Pentagon was proposing changes to correct the flaws.

While it is not improper for military contractors to use consultants in performing work for the Pentagon, the work must directly benefit the military if it is to be paid for by the Defense Department. Often, Pentagon investigators discovered, this test is not met.

Broader Look at Consultants

The Justice Department's continuing criminal investigation has focused attention on consultants and their role in the designing and selling of weapons, and the Defense Department has been criticized for using consultants too freely. Now the Pentagon's own inves-

Registration Off Since 1984 Vote

There has been a pronounced decline in the percentage of eligible Americans who are registered to vote, a research group reports.

Nationally, the percentage of eligible Americans who are registered is estimated to be 70.3 percent, down 2.2 points from the 1984 level.

The group's study concluded that in many of the 30 states where final figures are available the decline was among

Gov. Michael S. Dukakis having his picture taken a 10-year-old fan at a town meeting in Fairless Hi Pa., during a tour of the Northeast in which he e phasized the drug problem. Page A19. Vice Pr

Registration Off Since 1984 Vote

There has been a pronounced decline in the percentage of eligible Americans who are registered to vote, a research group reports.

Nationally, the percentage of eligible Americans who are registered is estimated to be 70.5 percent, down 2.2 points from the 1984 level.

The group's study concluded that in many of the 30 states where final figures are available the decline was among

PENTAGON REPORTS IMPROPER CHARGES FOR CONSULTANTS

CONTRACTORS CRITICIZED

Inquiry Shows Routine Billing of Government by Industry on Fees, Some Dubious

By JOHN H. CUSHMAN Jr.
Special to the New York Times

WASHINGTON, Nov. 3 — A Pentagon investigation has found that the nation's largest military contractors routinely charge the Defense Department for hundreds of millions of dollars paid to consultants, often without justification.

The report of the investigation said that neither the military's current rules nor the contractors' own policies are adequate to assure that the Government does not improperly pay for privately arranged consulting work. Senior Defense Department officials said the Pentagon was proposing changes to correct the flaws.

While it is not improper for military contractors to use consultants in performing work for the Pentagon, the work must directly benefit the military if it is to be paid for by the Defense Department. Often, Pentagon investigators discovered, this test is not met.

Broader Look at Consultants

The Justice Department's continuing criminal investigation has focused attention on consultants and their role in the designing and selling of weapons, and the Defense Department has been criticized for using consultants too freely. Now the Pentagon's own inves-

Memory from the Stack vs. Heap

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

- Until today, 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

Cleaning Up

- When declaring local variables or parameters, C++ will automatically handle memory allocation and deallocation for you.

Cleaning Up

- When declaring local variables or parameters, C++ will automatically handle memory allocation and deallocation for you.
  - Memory allocation is the process by which the computer hands you a piece of computer memory in which you can store data.

Cleaning Up

- When declaring local variables or parameters, C++ will automatically handle memory allocation and deallocation for you.
    - Memory allocation is the process by which the computer hands you a piece of computer memory in which you can store data.
    - Memory deallocation is the process by which control of this memory (data storage location) is relinquished back to the computer

Cleaning Up

- When declaring local variables or parameters, C++ will automatically handle memory allocation and deallocation for you.

- When using **`new`**, you are responsible for deallocating the memory you allocate.

Cleaning Up

- When declaring local variables or parameters, C++ will automatically handle memory allocation and deallocation for you.

- When using **`new`**, you are responsible for deallocating the memory you allocate.

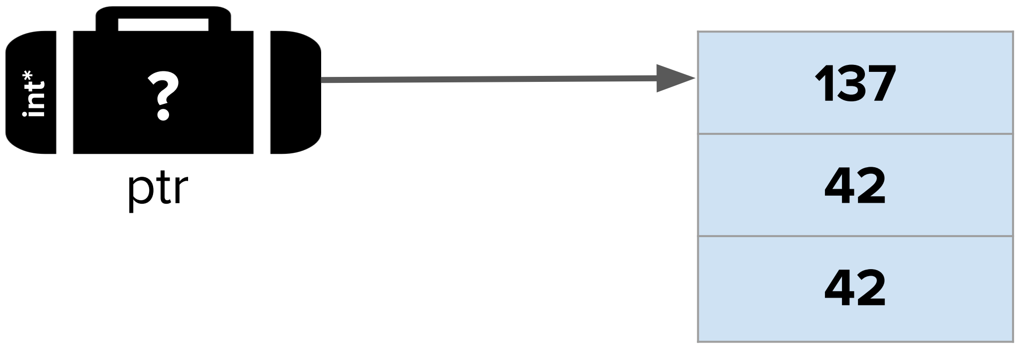
- If you don't, you get a **memory leak**. Your program will never be able to use that memory again.
  - Too many leaks can cause a program to crash – it's important to not leak memory!

Cleaning Up

- You can deallocate (free) memory with the **delete[]** operator:

```
delete[] ptr;
```

- This destroys the array pointed to by the given pointer, not the pointer itself.
  - You can think of this operation as relinquishing control over the memory back to the computer.

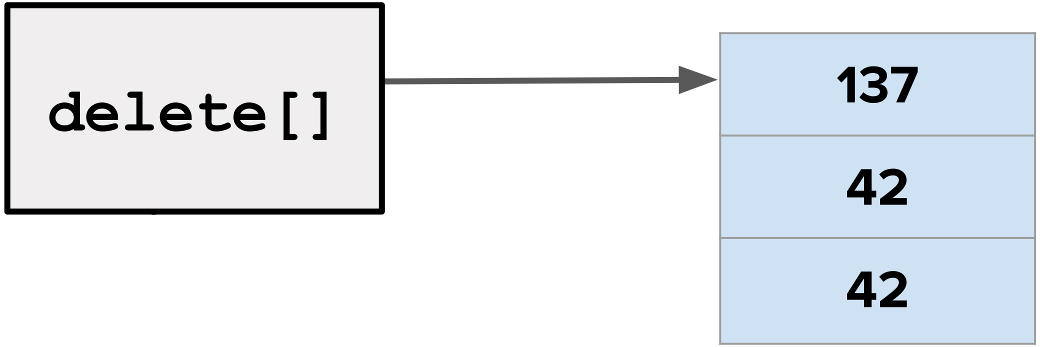


Cleaning Up

- You can deallocate (free) memory with the `delete[]` operator:

```
delete[] ptr;
```

- This destroys the array pointed to by the given pointer, not the pointer itself.
    - You can think of this operation as relinquishing control over the memory back to the computer.

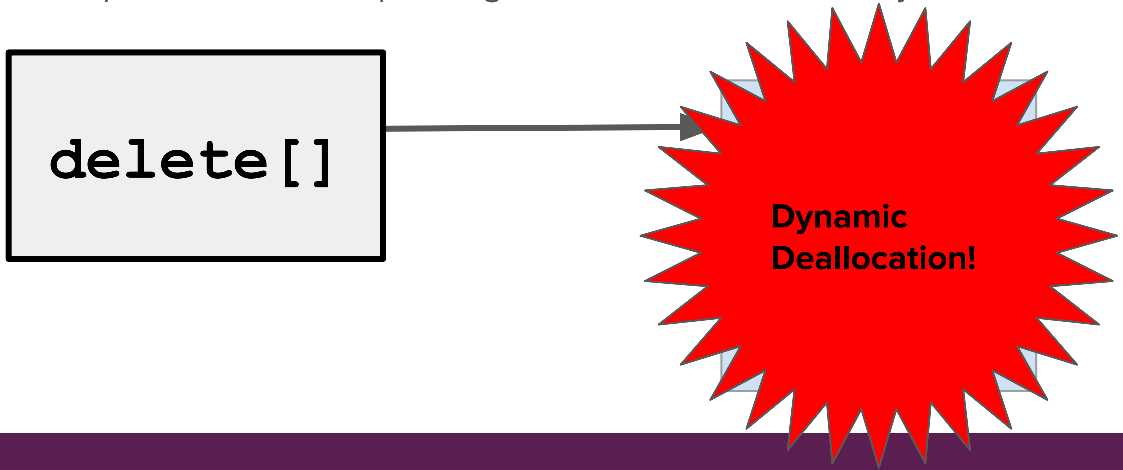


Cleaning Up

- You can deallocate (free) memory with the `delete[]` operator:

```
delete[] ptr;
```

- This destroys the array pointed to by the given pointer, not the pointer itself.
  - You can think of this operation as relinquishing control over the memory back to the computer.

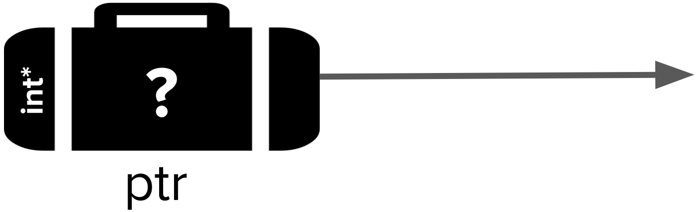


Cleaning Up

- You can deallocate (free) memory with the `delete[]` operator:

```
delete[] ptr;
```

- This destroys the array pointed to by the given pointer, not the pointer itself.
  - You can think of this operation as relinquishing control over the memory back to the computer.

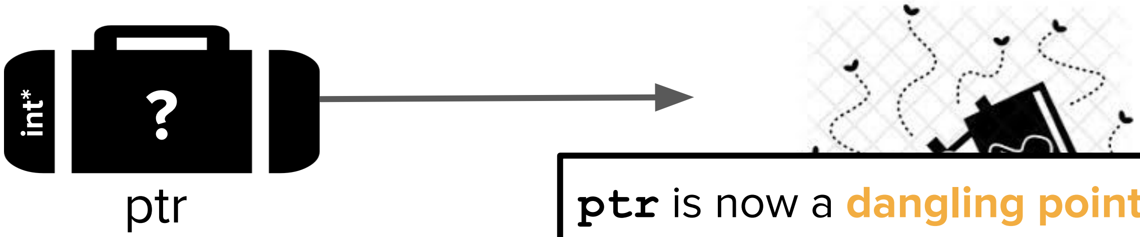


Cleaning Up

- You can deallocate (free) memory with the `delete[]` operator:

```
delete[] ptr;
```

- This destroys the array pointed to by the given pointer, not the pointer itself.
  - You can think of this operation as relinquishing control over the memory back to the computer.



`ptr` is now a **dangling pointer**. We can re-assign it to point somewhere else, but if we try to read from it or write to it, very bad, bad things will happen!

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. (This is why we didn't teach arrays earlier!)
- You are responsible for freeing any memory you explicitly allocate by calling **`delete[]`**.
- Once you've deleted the memory pointed at by a pointer, you have a dangling pointer and shouldn't read or write from it.

Overflow: Designing OurVector

Arrays vs. Vectors – A Common Mistake

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

Breakout Activity:
`OurVector` class design

Summary

Dynamic Memory and Arrays

- We've learned about **classes**, which have an **interface** and **implementation**.

Dynamic Memory and Arrays

- We’ve learned about **classes**, which have an **interface** and **implementation**.

- When implementing classes at the *lowest level of abstraction*, we need to use **dynamic memory** as a fundamental building block for specifying how much memory something needs.
  - We use the keyword **new** to allocate dynamic memory.
  - We keep track of that memory with a **pointer**. (more on pointers next week!)
  - We must clean up the memory when we’re done with **delete**.

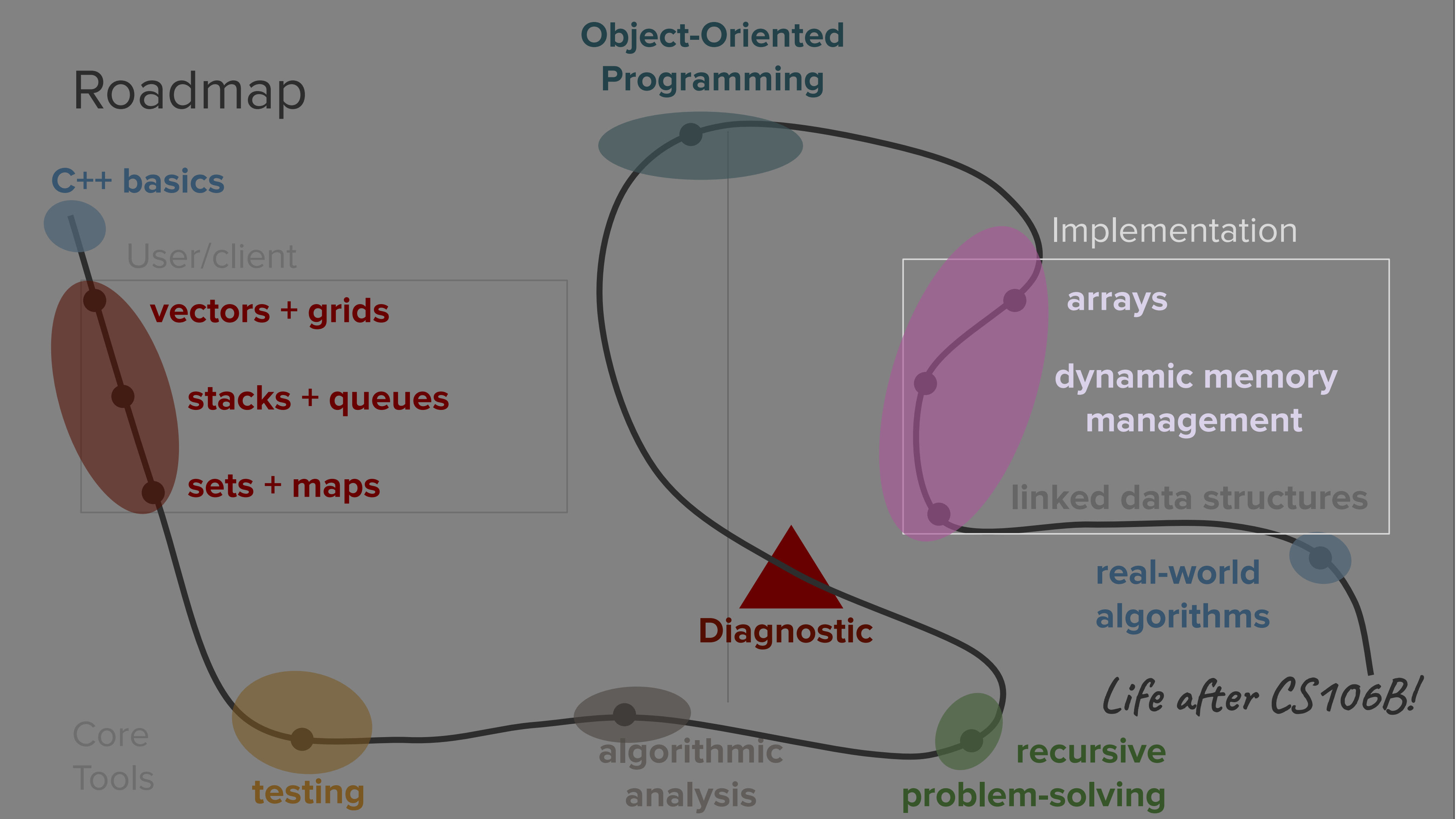
Dynamic Memory and Arrays

- We’ve learned about **classes**, which have an **interface** and **implementation**.

- When implementing classes at the *lowest level of abstraction*, we need to use **dynamic memory** as a fundamental building block for specifying how much memory something needs.
    - We use the keyword **new** to allocate dynamic memory.
    - We keep track of that memory with a **pointer**. (more on pointers next week!)
    - We must clean up the memory when we’re done with **delete**.

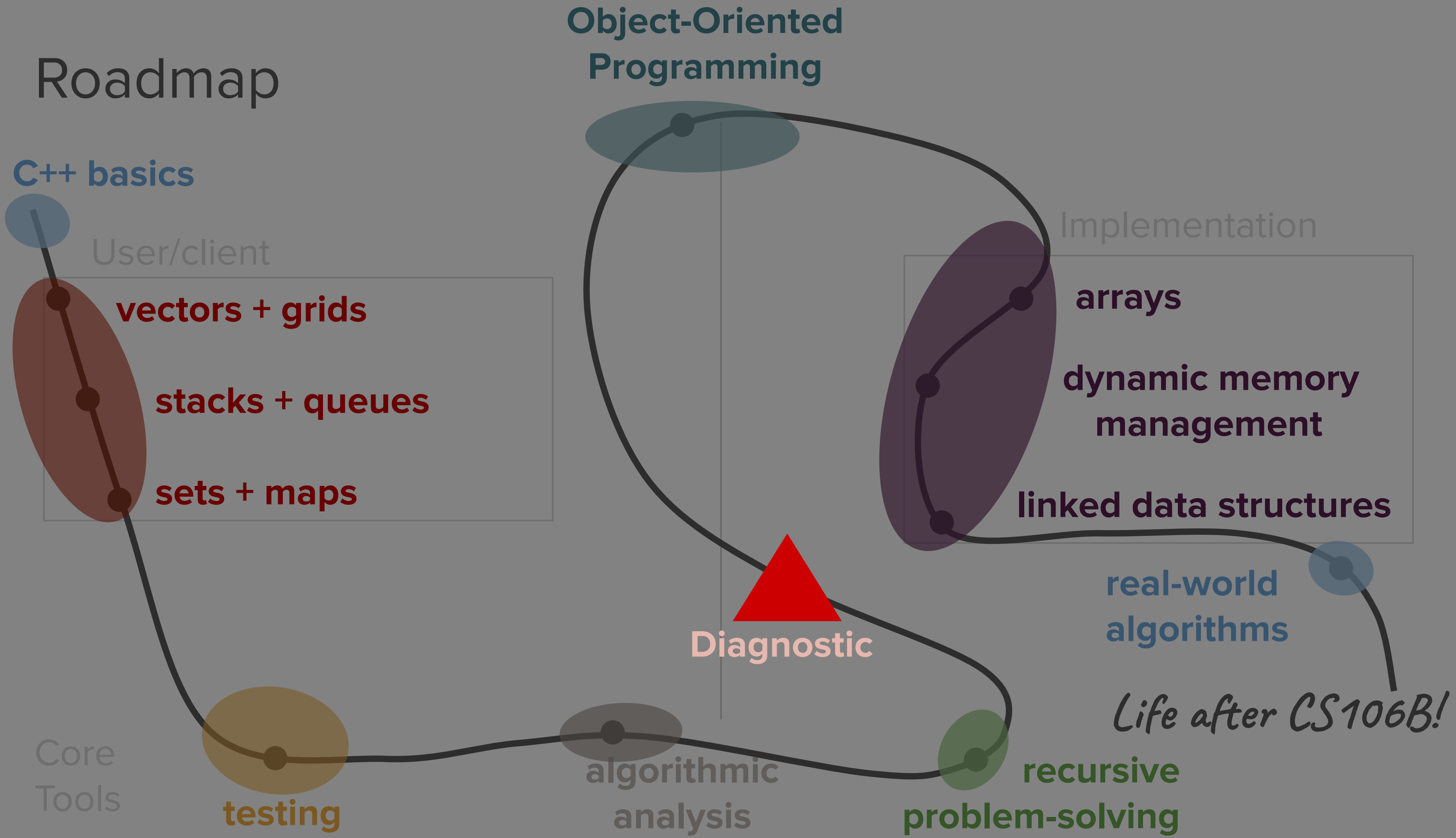
- So far, we’ve learned how to allocate dynamic memory using **arrays**, which give us a contiguous block of memory that all stores one particular type (int, string, double, etc.).

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

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

Implementing a Dynamic ADT