Dynamic Memory and Arrays

What are real-world examples of classes and abstractions?

(put your answers in the chat)
Roadmap

Object-Oriented Programming

C++ basics

User/client

vectors + grids

stacks + queues

sets + maps

arrays

dynamic memory management

linked data structures

real-world algorithms

recursive problem-solving

Life after CS106B!

Core Tools

testing

algorithmic analysis

Implementation

Diagnostic
Roadmap

C++ basics
- User/client
  - vectors + grids
  - stacks + queues
  - sets + maps

Object-Oriented Programming
- Implementation
  - arrays
  - dynamic memory management
  - linked data structures

Core Tools
- testing
- algorithmic analysis

Life after CS106B!
- 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
abstraction
Design that hides the details of how something works while still allowing the user to access complex functionality.

How do we accomplish this in C++? With classes!
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: *Vectors*, *Maps*, *Stacks*, *Queues*

- 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?

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

- Public member variables declared in the header file are automatically accessible in the `.cpp` file.

- 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 tomorrow!)
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

● The **mid-quarter diagnostic** will be released later tonight!
  ○ The link to access your personalized diagnostic access portal will be posted on the homepage of the website tonight at 12:01am PDT Friday and will remain up until 11:59pm PDT Sunday.
  ○ Do not visit this link until you are ready to complete the diagnostic.
  ○ We are logging download and submission times – you must download and submit the diagnostic within a 3-hour time span.

● Assignment 3 is due tonight, **Thursday, July 16 at 11:59pm**.

● Trip is hosting a diagnostic review session **tonight at 7pm PDT**.

● Revisions for Assignment 2 are now available.
Words of Advice

● Best of luck on the diagnostic! We hope that you all rock it!

● This is chance to demonstrate how much you've learned in just 3 weeks. The purpose of the diagnostic is truly "diagnostic" – to help you self-assess your own areas of strength and areas of potential growth. We expect everyone to have areas of improvement!

● Make sure to collect the resources that you plan to use in advance.

● Get a good night's sleep, eat a solid meal, get some exercise, and rock the diagnostic!
Where are we now?
Abstract data structures (vectors, maps, etc.)

arrays
- dynamic memory management
- linked data structures

testing

algorithmic analysis

recursive problem-solving
classes
object-oriented programming

testing
algorithmic analysis
recursive problem-solving

abstract data structures
(vectors, maps, etc.)

arrays
dynamic memory management
linked data structures
We've now crossed the abstraction boundary!

- arrays
- dynamic memory management
- linked data structures
RandomBag Revisited
#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.

<table>
<thead>
<tr>
<th>Index: 0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>
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. \( \text{int}^* \), \( \text{string}^* \), \( \text{Vector<double>}^* \)
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.
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];
  ```
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. \( \text{int}^* \), \( \text{string}^* \), \( \text{Vector}<\text{double}>^* \)
- Then, create a new array with the `new` keyword and assign the pointer to point to it.
- In two separate steps:
  ```cpp
  T* arr;
  arr = new T[size];
  ```
- Or, in the same line:
  ```cpp
  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 like the specific coordinates of where a piece of memory exists on the computer.
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.
Dynamic Allocation Demo
```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;
    }
}
```
```c++
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;
    }
}

numValues = 7
```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;
    }
}

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.
```c++
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;
    }
}
```
```c++
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;
    }
}
```

We

<table>
<thead>
<tr>
<th>Index:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>numValues</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr</td>
<td>0x8084ffff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```c++
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;
    }
}
```c++
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;
    }
    return 0;
}
```

We Can

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```c
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

<table>
<thead>
<tr>
<th>Index</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>numValues</td>
<td>7</td>
</tr>
<tr>
<td>arr</td>
<td>0x8084ffff</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
</tr>
</tbody>
</table>
```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;
    }
}
```
```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;
    }
}
```

```
We Can Dance
```

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>numValues</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr</td>
<td>0x8084ffff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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;
    }
}

numValues
arr
i

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>We</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Can</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x8084fff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```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 = getInt("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;
    }
}

Index:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>We</td>
<td>Can</td>
<td>Dance</td>
<td>If</td>
<td>We</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

numValues = 7
arr = 0x8084ffff
i = 4
```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;
    }
}
```
```c++
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;
    }
}
```
```c++
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;
    }
}

We Can Dance If We Want

Index: 0 1 2 3 4 5 6
```c++
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 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;
}
```c++
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;
    }
}
```
```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;
    }
}
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>We</td>
</tr>
<tr>
<td>1</td>
<td>Can</td>
</tr>
<tr>
<td>2</td>
<td>Dance</td>
</tr>
<tr>
<td>3</td>
<td>If</td>
</tr>
<tr>
<td>4</td>
<td>We</td>
</tr>
<tr>
<td>5</td>
<td>Want</td>
</tr>
<tr>
<td>6</td>
<td>To</td>
</tr>
</tbody>
</table>

```
numValues
arr
```

Index: 0 1 2 3 4 5 6 7
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

numValues: 7
arr: 0x8084ffff
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.”
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.”
- 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...
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.

Notably, the percentage of registered Americans is estimated to be 18.5 percent, down 23 points from the 1984 level.

The group's study concluded that in many of the 36 states where final figures are available, the decline was among Republicans, Democrats, and independents.

The report, which is based on a survey of 1,200 registered voters, found that the percentage of registered voters in the 18- to 29-year-old age group has dropped from 28.6 percent in 1984 to 17.8 percent in 2018.

The study also found that the percentage of registered voters who are black has dropped from 40.1 percent in 1984 to 34.1 percent in 2018.

In contrast, the percentage of registered voters who are white has increased from 75.6 percent in 1984 to 79.1 percent in 2018.

The study suggests that the decline in registration is due to a number of factors, including the increasing use of technology and the decreasing importance of the political process among young people.

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.

Notably, the percentage of registered Americans is estimated to be 18.5 percent, down 23 points from the 1984 level.

The group's study concluded that in many of the 36 states where final figures are available, the decline was among Republicans, Democrats, and independents.

The report, which is based on a survey of 1,200 registered voters, found that the percentage of registered voters in the 18- to 29-year-old age group has dropped from 28.6 percent in 1984 to 17.8 percent in 2018.

The study also found that the percentage of registered voters who are black has dropped from 40.1 percent in 1984 to 34.1 percent in 2018.

In contrast, the percentage of registered voters who are white has increased from 75.6 percent in 1984 to 79.1 percent in 2018.

The study suggests that the decline in registration is due to a number of factors, including the increasing use of technology and the decreasing importance of the political process among young people.

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.

Notably, the percentage of registered Americans is estimated to be 18.5 percent, down 23 points from the 1984 level.

The group's study concluded that in many of the 36 states where final figures are available, the decline was among Republicans, Democrats, and independents.

The report, which is based on a survey of 1,200 registered voters, found that the percentage of registered voters in the 18- to 29-year-old age group has dropped from 28.6 percent in 1984 to 17.8 percent in 2018.

The study also found that the percentage of registered voters who are black has dropped from 40.1 percent in 1984 to 34.1 percent in 2018.

In contrast, the percentage of registered voters who are white has increased from 75.6 percent in 1984 to 79.1 percent in 2018.

The study suggests that the decline in registration is due to a number of factors, including the increasing use of technology and the decreasing importance of the political process among young people.

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.

Notably, the percentage of registered Americans is estimated to be 18.5 percent, down 23 points from the 1984 level.

The group's study concluded that in many of the 36 states where final figures are available, the decline was among Republicans, Democrats, and independents.

The report, which is based on a survey of 1,200 registered voters, found that the percentage of registered voters in the 18- to 29-year-old age group has dropped from 28.6 percent in 1984 to 17.8 percent in 2018.

The study also found that the percentage of registered voters who are black has dropped from 40.1 percent in 1984 to 34.1 percent in 2018.

In contrast, the percentage of registered voters who are white has increased from 75.6 percent in 1984 to 79.1 percent in 2018.

The study suggests that the decline in registration is due to a number of factors, including the increasing use of technology and the decreasing importance of the political process among young people.

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.

Notably, the percentage of registered Americans is estimated to be 18.5 percent, down 23 points from the 1984 level.

The group's study concluded that in many of the 36 states where final figures are available, the decline was among Republicans, Democrats, and independents.

The report, which is based on a survey of 1,200 registered voters, found that the percentage of registered voters in the 18- to 29-year-old age group has dropped from 28.6 percent in 1984 to 17.8 percent in 2018.

The study also found that the percentage of registered voters who are black has dropped from 40.1 percent in 1984 to 34.1 percent in 2018.

In contrast, the percentage of registered voters who are white has increased from 75.6 percent in 1984 to 79.1 percent in 2018.

The study suggests that the decline in registration is due to a number of factors, including the increasing use of technology and the decreasing importance of the political process among young people.
JUDGMENT DAY
The Sentencing of Robert Morris Jr.
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:

  ```cpp
  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:

  ```c
  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:

  \[
  \text{delete[]} \\ \text{ptr};
  \]

- 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.

● 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.
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:

```java
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

C++ basics

User/client

vectors + grids

stacks + queues

sets + maps

Object-Oriented Programming

Implementation

arrays

dynamic memory management

linked data structures

Real-world algorithms

Recursive problem-solving

Algorithmic analysis

Testing

Core tools

Life after CS106B!
Roadmap

C++ basics
- vectors + grids
- stacks + queues
- sets + maps

Object-Oriented Programming
- arrays
- dynamic memory management
- linked data structures

Implementation
- real-world algorithms
- recursive problem-solving

Diagnostic

Life after CS106B!

Core Tools
- testing
- algorithmic analysis
Implementing a Dynamic ADT