

Algorithms and Analysis

Lesson 5: *Use Arrays*



Variable length arrays, implementing stacks

Outline

1. **Why Arrays?**
2. Variable Length Arrays
3. Programming Language
4. Implementing Stacks



Use Arrays

- An array is a contiguous chunk of memory
- In C++ we can create arrays using
`int *array = new int[20]`
- The array has an access time of $\Theta(1)$
- The constant factor is small (i.e. access time ≈ 1 time step)
- Arrays provide a very efficient use of memory
- 95% of the time using arrays is going to give you the best performance

Use Arrays

- An array is a contiguous chunk of memory
- In C++ we can create arrays using
`int *array = new int[20]`
- The array has an access time of $\Theta(1)$
- The constant factor is small (i.e. access time ≈ 1 time step)
- Arrays provide a very efficient use of memory
- 95% of the time using arrays is going to give you the best performance

Use Arrays

- An array is a contiguous chunk of memory
- In C++ we can create arrays using
`int *array = new int[20]`
- The array has an access time of $\Theta(1)$
- The constant factor is small (i.e. access time ≈ 1 time step)
- Arrays provide a very efficient use of memory
- 95% of the time using arrays is going to give you the best performance

Use Arrays

- An array is a contiguous chunk of memory
- In C++ we can create arrays using
`int *array = new int[20]`
- The array has an access time of $\Theta(1)$
- The constant factor is small (i.e. access time ≈ 1 time step)
- Arrays provide a very efficient use of memory
- 95% of the time using arrays is going to give you the best performance

Use Arrays

- An array is a contiguous chunk of memory
- In C++ we can create arrays using
`int *array = new int[20]`
- The array has an access time of $\Theta(1)$
- The constant factor is small (i.e. access time ≈ 1 time step)
- Arrays provide a very efficient use of memory
- 95% of the time using arrays is going to give you the best performance

Use Arrays

- An array is a contiguous chunk of memory
- In C++ we can create arrays using
`int *array = new int[20]`
- The array has an access time of $\Theta(1)$
- The constant factor is small (i.e. access time ≈ 1 time step)
- Arrays provide a very efficient use of memory
- 95% of the time using arrays is going to give you the best performance

Use Arrays

- An array is a contiguous chunk of memory
- In C++ we can create arrays using
`int *array = new int[20]`
- The array has an access time of $\Theta(1)$
- The constant factor is small (i.e. access time ≈ 1 time step)
- Arrays provide a very efficient use of memory
- 95% of the time using arrays is going to give you the best performance, **although never use raw arrays!**

Disadvantages of Arrays

- Arrays have a fixed length
- Very often we don't know how big an array we want
 - ★ E.g. reading words from a file
- Adding or deleting elements from the middle of an array is costly
- Sorted arrays are expensive to maintain
- Arrays don't know how big they are

Disadvantages of Arrays

- Arrays have a fixed length
- Very often we don't know how big an array we want
 - ★ E.g. reading words from a file
- Adding or deleting elements from the middle of an array is costly
- Sorted arrays are expensive to maintain
- Arrays don't know how big they are

Disadvantages of Arrays

- Arrays have a fixed length
- Very often we don't know how big an array we want
 - ★ E.g. reading words from a file
- Adding or deleting elements from the middle of an array is costly
- Sorted arrays are expensive to maintain
- Arrays don't know how big they are

Disadvantages of Arrays

- Arrays have a fixed length
- Very often we don't know how big an array we want
 - ★ E.g. reading words from a file
- Adding or deleting elements from the middle of an array is costly
- Sorted arrays are expensive to maintain
- Arrays don't know how big they are

Disadvantages of Arrays

- Arrays have a fixed length
- Very often we don't know how big an array we want
 - ★ E.g. reading words from a file
- Adding or deleting elements from the middle of an array is costly
- Sorted arrays are expensive to maintain
- Arrays don't know how big they are

Disadvantages of Arrays

- Arrays have a fixed length
- Very often we don't know how big an array we want
 - ★ E.g. reading words from a file
- Adding or deleting elements from the middle of an array is costly
- Sorted arrays are expensive to maintain
- Arrays don't know how big they are—**annoying**

Outline

1. Why Arrays?
2. **Variable Length Arrays**
3. Programming Language
4. Implementing Stacks



Variable Length Arrays

- We want a variable length array
- Initially a variable length array would have length zero
- We should be able to
 - ★ Add an element to an array
 - ★ Access any element in the array
 - ★ Change an element
 - ★ Delete elements
 - ★ Know how many elements we have

Variable Length Arrays

- We want a variable length array
- Initially a variable length array would have length zero
- We should be able to
 - ★ Add an element to an array
 - ★ Access any element in the array
 - ★ Change an element
 - ★ Delete elements
 - ★ Know how many elements we have

Variable Length Arrays

- We want a variable length array
- Initially a variable length array would have length zero
- We should be able to
 - ★ Add an element to an array
 - ★ Access any element in the array
 - ★ Change an element
 - ★ Delete elements
 - ★ Know how many elements we have

ADT for a List

- What do we want of a list of `ints`?

ADT for a List

- What do we want of a list of **ints**?
 - ★ **void** push_back(**int** value)

ADT for a List

- What do we want of a list of **ints**?
 - ★ **void** `push_back(int value)`
 - ★ random access `array[i]`

ADT for a List

- What do we want of a list of **ints**?
 - ★ **void** push_back(**int** value)
 - ★ random access `array[i]`
 - ★ **int** size()

ADT for a List

- What do we want of a list of **ints**?
 - ★ **void** push_back(**int** value)
 - ★ random access `array[i]`
 - ★ **int** size()
- It would be useful if it resized

ADT for a List

- What do we want of a list of **ints**?
 - ★ **void** push_back(**int** value)
 - ★ random access `array[i]`
 - ★ **int** size()
- It would be useful if it resized
- It would be great to have some algorithms (e.g. sort) that can be run on a list

Implementation

- How should we implement a list?

Implementation

- How should we implement a list?
- Use an array, of course!

Implementation

- How should we implement a list?
- Use an array, of course!
- We need to distinguish between
 - ★ the number of elements in the list `size()`
 - ★ the number of elements in the array `capacity()`

Implementation

- How should we implement a list?
- Use an array, of course!
- We need to distinguish between
 - ★ the number of elements in the list `size()`
 - ★ the number of elements in the array `capacity()`
- If the number of elements grows larger than the capacity then we need to increase the capacity

Initial Capacity

- We could prevent resizing arrays by using a huge initial capacity
- However, how big is big enough?
- What happens when we have an array of arrays?
- Memory like time is resource we should care about
- In an analogy with **time complexity** we also care about **space complexity** (i.e. how much memory we need)
- If we want to store n elements it is reasonable to expect that we use cn bits of memory where we want to keep c small

Initial Capacity

- We could prevent resizing arrays by using a huge initial capacity
- However, how big is big enough?
- What happens when we have an array of arrays?
- Memory like time is resource we should care about
- In an analogy with **time complexity** we also care about **space complexity** (i.e. how much memory we need)
- If we want to store n elements it is reasonable to expect that we use cn bits of memory where we want to keep c small

Initial Capacity

- We could prevent resizing arrays by using a huge initial capacity
- However, how big is big enough?
- What happens when we have an array of arrays?
- Memory like time is resource we should care about
- In an analogy with **time complexity** we also care about **space complexity** (i.e. how much memory we need)
- If we want to store n elements it is reasonable to expect that we use cn bits of memory where we want to keep c small

Initial Capacity

- We could prevent resizing arrays by using a huge initial capacity
- However, how big is big enough?
- What happens when we have an array of arrays?
- Memory like time is resource we should care about
- In an analogy with **time complexity** we also care about **space complexity** (i.e. how much memory we need)
- If we want to store n elements it is reasonable to expect that we use cn bits of memory where we want to keep c small

Initial Capacity

- We could prevent resizing arrays by using a huge initial capacity
- However, how big is big enough?
- What happens when we have an array of arrays?
- Memory like time is resource we should care about
- In an analogy with **time complexity** we also care about **space complexity** (i.e. how much memory we need)
- If we want to store n elements it is reasonable to expect that we use cn bits of memory where we want to keep c small

Initial Capacity

- We could prevent resizing arrays by using a huge initial capacity
- However, how big is big enough?
- What happens when we have an array of arrays?
- Memory like time is resource we should care about
- In an analogy with **time complexity** we also care about **space complexity** (i.e. how much memory we need)
- If we want to store n elements it is reasonable to expect that we use cn bits of memory where we want to keep c small

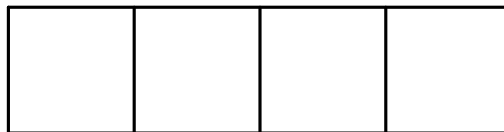
Resizing Memory

- We start with some reasonable capacity



Resizing Memory

- We start with some reasonable capacity
- We can add elements



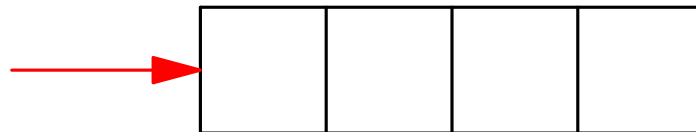
$\text{size}() = 0$

$\text{capacity} = 4$

Resizing Memory

- We start with some reasonable capacity
- We can add elements

`list.push_back(52)`



`size() = 0`

`capacity = 4`

Resizing Memory

- We start with some reasonable capacity
- We can add elements

`list.push_back(31)` →


52			
----	--	--	--

`size() = 1`

`capacity = 4`

Resizing Memory

- We start with some reasonable capacity
- We can add elements

`list.push_back(94)` 


52	31		
----	----	--	--

`size() = 2`

`capacity = 4`

Resizing Memory

- We start with some reasonable capacity
- We can add elements

`list.push_back(50)` 


52	31	94	
----	----	----	--

`size() = 3`

`capacity = 4`

Resizing Memory

- We start with some reasonable capacity
- We can add elements until we reach the capacity

`list.push_back(82)` 

52	31	94	50
----	----	----	----

`size() = 4`

`capacity = 4`

Resizing Memory

- We start with some reasonable capacity
- We can add elements until we reach the capacity
- A simple method for resizing memory is
 - ★ create a new array with double the capacity of the old array

`list.push_back(82)` →

52	31	94	50
----	----	----	----

`size() = 4`

`capacity = 8`



Resizing Memory

- We start with some reasonable capacity
- We can add elements until we reach the capacity
- A simple method for resizing memory is
 - ★ create a new array with double the capacity of the old array
 - ★ copy the existing elements from the old array to the new array

`list.push_back(82)` →

52	31	94	50
----	----	----	----

`size() = 4`

`capacity = 8`



Resizing Memory

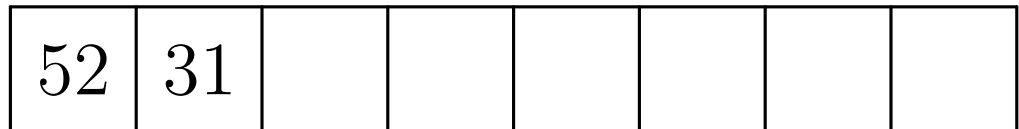
- We start with some reasonable capacity
- We can add elements until we reach the capacity
- A simple method for resizing memory is
 - ★ create a new array with double the capacity of the old array
 - ★ copy the existing elements from the old array to the new array

`list.push_back(82)` →

52	31	94	50
----	----	----	----

`size() = 4`

`capacity = 8`



Resizing Memory

- We start with some reasonable capacity
- We can add elements until we reach the capacity
- A simple method for resizing memory is
 - ★ create a new array with double the capacity of the old array
 - ★ copy the existing elements from the old array to the new array

`list.push_back(82)` →

52	31	94	50
----	----	----	----

`size() = 4`

`capacity = 8`

52	31	94					
----	----	----	--	--	--	--	--

Resizing Memory

- We start with some reasonable capacity
- We can add elements until we reach the capacity
- A simple method for resizing memory is
 - ★ create a new array with double the capacity of the old array
 - ★ copy the existing elements from the old array to the new array

`list.push_back(82)` →

52	31	94	50
----	----	----	----

`size() = 4`

`capacity = 8`

52	31	94	50				
----	----	----	----	--	--	--	--

Resizing Memory

- We start with some reasonable capacity
- We can add elements until we reach the capacity
- A simple method for resizing memory is
 - ★ create a new array with double the capacity of the old array
 - ★ copy the existing elements from the old array to the new array

`list.push_back(82)` 

`size() = 4`

`capacity = 8`

52	31	94	50				
----	----	----	----	--	--	--	--

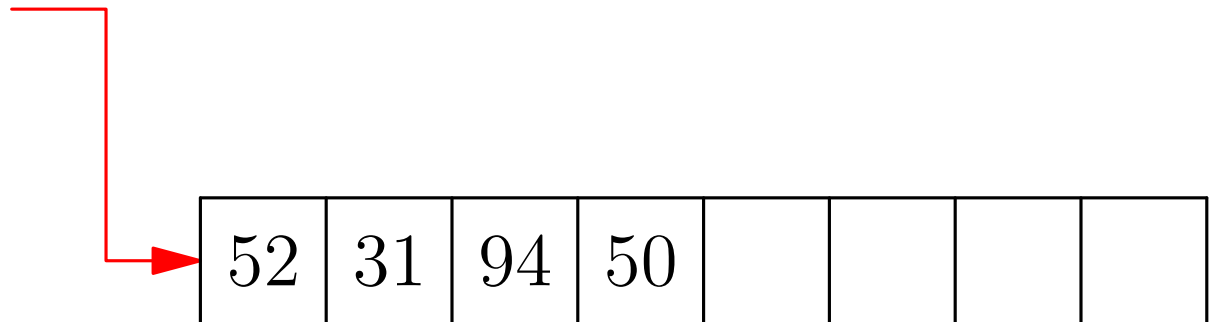
Resizing Memory

- We start with some reasonable capacity
- We can add elements until we reach the capacity
- A simple method for resizing memory is
 - ★ create a new array with double the capacity of the old array
 - ★ copy the existing elements from the old array to the new array

`list.push_back(82)`

`size() = 4`

`capacity = 8`



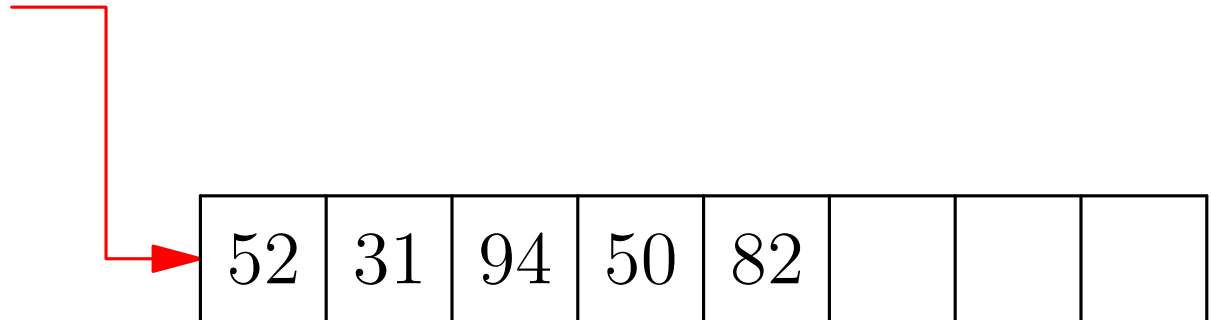
Resizing Memory

- We start with some reasonable capacity
- We can add elements until we reach the capacity
- A simple method for resizing memory is
 - ★ create a new array with double the capacity of the old array
 - ★ copy the existing elements from the old array to the new array

`list.push_back(82)`

`size() = 5`

`capacity = 8`



Amortised Time Analysis

- How efficient is resizing?
- Most `push_back(elem)` operations are $\Theta(1)$
- When we are at full capacity we have to copy all elements
- Adding to a full array is slow but it is **amortised** by other quick adds

amortised: effect of a single operations 'deadened' by other operations

Amortised Time Analysis

- How efficient is resizing?
- Most `push_back(elem)` operations are $\Theta(1)$
- When we are at full capacity we have to copy all elements
- Adding to a full array is slow but it is **amortised** by other quick adds

amortised: effect of a single operations 'deadened' by other operations

Amortised Time Analysis

- How efficient is resizing?
- Most `push_back(elem)` operations are $\Theta(1)$
- When we are at full capacity we have to copy all elements
- Adding to a full array is slow but it is **amortised** by other quick adds

amortised: effect of a single operations 'deadened' by other operations

Amortised Time Analysis

- How efficient is resizing?
- Most `push_back(elem)` operations are $\Theta(1)$
- When we are at full capacity we have to copy all elements
- Adding to a full array is slow but it is **amortised** by other quick adds

amortised: effect of a single operations 'deadened' by other operations

Amortised Time Analysis

- How efficient is resizing?
- Most `push_back(elem)` operations are $\Theta(1)$
- When we are at full capacity we have to copy all elements
- Adding to a full array is slow but it is **amortised** by other quick adds

amortised: effect of a single operations 'deadened' by other operations

Example

- If we have an initial capacity of 10 and add 100 elements then the number of operations needed is
 - ★ adds: 100

Example

- If we have an initial capacity of 10 and add 100 elements then the number of operations needed is
 - ★ adds: 100
 - ★ copies: 10

Example

- If we have an initial capacity of 10 and add 100 elements then the number of operations needed is
 - ★ adds: 100
 - ★ copies: $10+20$

Example

- If we have an initial capacity of 10 and add 100 elements then the number of operations needed is
 - ★ adds: 100
 - ★ copies: $10+20+40$

Example

- If we have an initial capacity of 10 and add 100 elements then the number of operations needed is
 - ★ adds: 100
 - ★ copies: $10+20+40+80$

Example

- If we have an initial capacity of 10 and add 100 elements then the number of operations needed is
 - ★ adds: 100
 - ★ copies: $10+20+40+80=150$

Example

- If we have an initial capacity of 10 and add 100 elements then the number of operations needed is
 - ★ adds: 100
 - ★ copies: $10+20+40+80=150$
 - ★ **new int []**: 4

Example

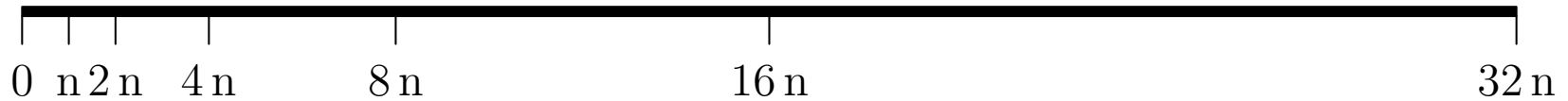
- If we have an initial capacity of 10 and add 100 elements then the number of operations needed is
 - ★ adds: 100
 - ★ copies: $10+20+40+80=150$
 - ★ **new int** []: 4
- 250 adds and copies operations + 4 **new** operations

General Time Analysis

- If we perform N adds with an initial capacity of n

General Time Analysis

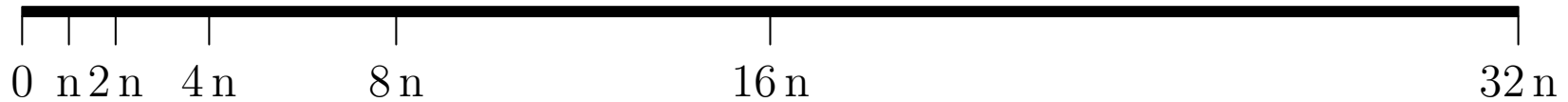
- If we perform N adds with an initial capacity of n
- We must perform m copies where



$$n \times 2^{m-1} < N \leq n \times 2^m$$

General Time Analysis

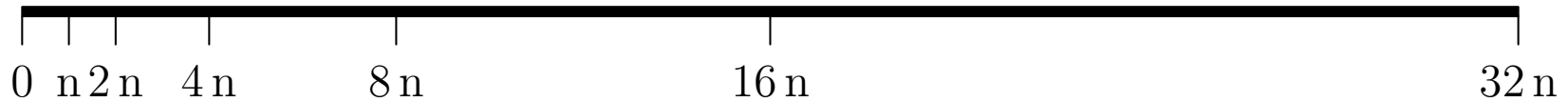
- If we perform N adds with an initial capacity of n
- We must perform m copies where



$$n \times 2^{m-1} < N \leq n \times 2^m \quad \text{i.e.} \quad m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

General Time Analysis

- If we perform N adds with an initial capacity of n
- We must perform m copies where



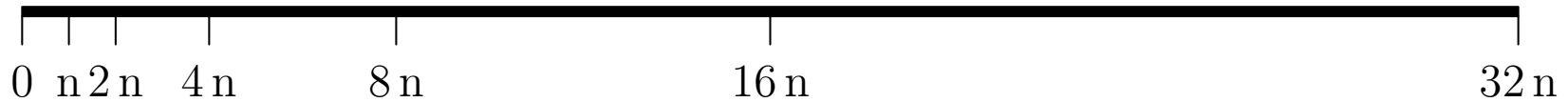
$$n \times 2^{m-1} < N \leq n \times 2^m \quad \text{i.e.} \quad m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

$$n + 2n + 4n + \dots + 2^{m-1}n$$

General Time Analysis

- If we perform N adds with an initial capacity of n
- We must perform m copies where



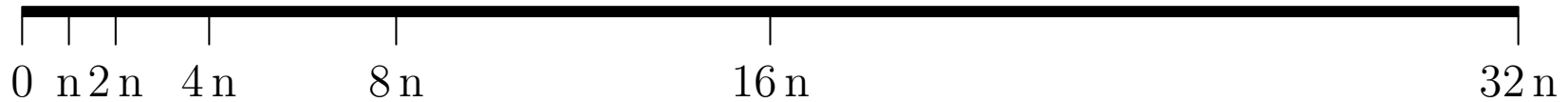
$$n \times 2^{m-1} < N \leq n \times 2^m \quad \text{i.e.} \quad m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

$$n + 2n + 4n + \cdots + 2^{m-1}n = n(1 + 2 + \cdots + 2^{m-1})$$

General Time Analysis

- If we perform N adds with an initial capacity of n
- We must perform m copies where



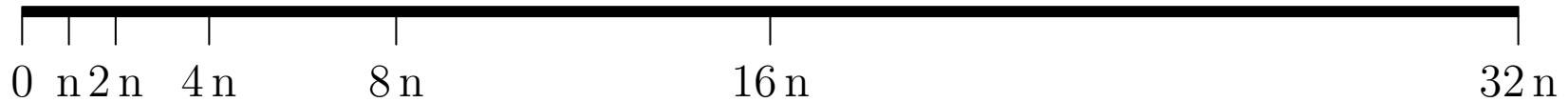
$$n \times 2^{m-1} < N \leq n \times 2^m \quad \text{i.e.} \quad m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

$$n + 2n + 4n + \cdots + 2^{m-1}n = n(1 + 2 + \cdots + 2^{m-1}) = n(2^m - 1)$$

General Time Analysis

- If we perform N adds with an initial capacity of n
- We must perform m copies where



$$n \times 2^{m-1} < N \leq n \times 2^m \quad \text{i.e.} \quad m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

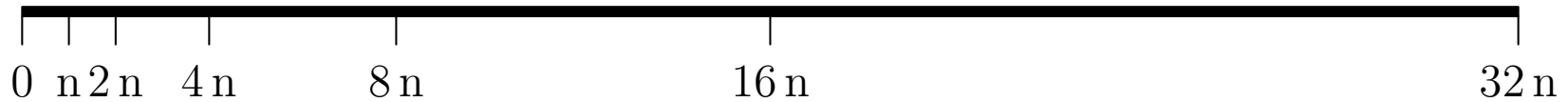
$$n + 2n + 4n + \dots + 2^{m-1}n = n(1 + 2 + \dots + 2^{m-1}) = n(2^m - 1)$$

- Total number of operations is

$$N + n(2^m - 1)$$

General Time Analysis

- If we perform N adds with an initial capacity of n
- We must perform m copies where



$$n \times 2^{m-1} < N \leq n \times 2^m \quad \text{i.e.} \quad m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

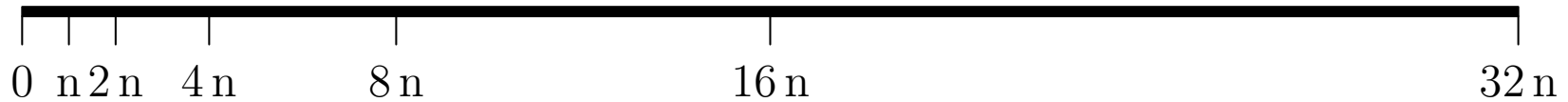
$$n + 2n + 4n + \dots + 2^{m-1}n = n(1 + 2 + \dots + 2^{m-1}) = n(2^m - 1)$$

- Total number of operations is

$$N + n(2^m - 1) = N + n2^{\left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil} - n$$

General Time Analysis

- If we perform N adds with an initial capacity of n
- We must perform m copies where



$$n \times 2^{m-1} < N \leq n \times 2^m \quad \text{i.e.} \quad m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

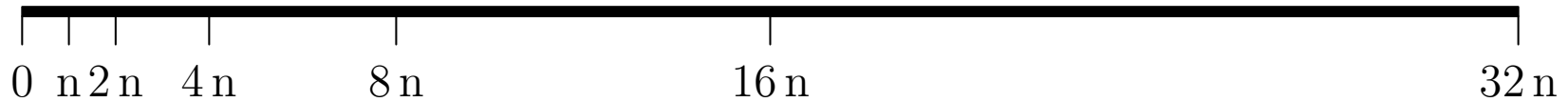
$$n + 2n + 4n + \dots + 2^{m-1}n = n(1 + 2 + \dots + 2^{m-1}) = n(2^m - 1)$$

- Total number of operations is (using $\lceil \log(a) \rceil < \log(a) + 1$)

$$N + n(2^m - 1) = N + n2^{\lceil \log_2(\frac{N}{n}) \rceil} - n < N + 2N - n$$

General Time Analysis

- If we perform N adds with an initial capacity of n
- We must perform m copies where



$$n \times 2^{m-1} < N \leq n \times 2^m \quad \text{i.e.} \quad m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

$$n + 2n + 4n + \cdots + 2^{m-1}n = n(1 + 2 + \cdots + 2^{m-1}) = n(2^m - 1)$$

- Total number of operations is (using $\lceil \log(a) \rceil < \log(a) + 1$)

$$N + n(2^m - 1) = N + n2^{\lceil \log_2(\frac{N}{n}) \rceil} - n < N + 2N - n < 3N$$

Insertion and Deletion

- In C++ `vector<T>` is a resizable array and used a lot
- But if you try to insert or delete an element anywhere other than the end then you have to shove all the subsequent elements one space forward
- This is not the right data structure if you want to keep elements in order
- Linked lists allow you to splice in a sublist into a list in constant time

Insertion and Deletion

- In C++ `vector<T>` is a resizable array and used a lot
- But if you try to insert or delete an element anywhere other than the end then you have to shove all the subsequent elements one space forward
- This is not the right data structure if you want to keep elements in order
- Linked lists allow you to splice in a sublist into a list in constant time

Insertion and Deletion

- In C++ `vector<T>` is a resizable array and used a lot
- But if you try to insert or delete an element anywhere other than the end then you have to shove all the subsequent elements one space forward
- This is not the right data structure if you want to keep elements in order
- Linked lists allow you to splice in a sublist into a list in constant time

Insertion and Deletion

- In C++ `vector<T>` is a resizable array and used a lot
- But if you try to insert or delete an element anywhere other than the end then you have to shove all the subsequent elements one space forward
- This is not the right data structure if you want to keep elements in order (binary trees will do that for you much more efficiently)
- Linked lists allow you to splice in a sublist into a list in constant time

Insertion and Deletion

- In C++ `vector<T>` is a resizable array and used a lot
- But if you try to insert or delete an element anywhere other than the end then you have to shove all the subsequent elements one space forward
- This is not the right data structure if you want to keep elements in order (binary trees will do that for you much more efficiently)
- Linked lists allow you to splice in a sublist into a list in constant time

Insertion and Deletion

- In C++ `vector<T>` is a resizable array and used a lot
- But if you try to insert or delete an element anywhere other than the end then you have to shove all the subsequent elements one space forward
- This is not the right data structure if you want to keep elements in order (binary trees will do that for you much more efficiently)
- Linked lists allow you to splice in a sublist into a list in constant time **although linked lists have a lot of drawbacks**

Outline

1. Why Arrays?
2. Variable Length Arrays
3. **Programming Language**
4. Implementing Stacks



Computer Languages

- Different computer languages are designed for different roles and have different advantages and disadvantages
- **C++** was designed to be fast (as fast as C)
- **Java** was designed to be very safe (avoiding lots of bugs)
- **Python** was designed so you can rapidly write powerful programmes with a small amount of code

Computer Languages

- Different computer languages are designed for different roles and have different advantages and disadvantages
- **C++** was designed to be fast (as fast as C)
- **Java** was designed to be very safe (avoiding lots of bugs)
- **Python** was designed so you can rapidly write powerful programmes with a small amount of code

Computer Languages

- Different computer languages are designed for different roles and have different advantages and disadvantages
- **C++** was designed to be fast (as fast as C), it pays the price of allowing bugs that hard to detect
- **Java** was designed to be vary safe (avoiding lots of bugs)
- **Python** was designed so you can rapidly write powerful programmes with a small amount of code

Computer Languages

- Different computer languages are designed for different roles and have different advantages and disadvantages
- **C++** was designed to be fast (as fast as C), it pays the price of allowing bugs that hard to detect
- **Java** was designed to be very safe (avoiding lots of bugs)
- **Python** was designed so you can rapidly write powerful programmes with a small amount of code

Computer Languages

- Different computer languages are designed for different roles and have different advantages and disadvantages
- **C++** was designed to be fast (as fast as C), it pays the price of allowing bugs that hard to detect
- **Java** was designed to be vary safe (avoiding lots of bugs), **but is not fast and a bit long winded**
- **Python** was designed so you can rapidly write powerful programmes with a small amount of code

Computer Languages

- Different computer languages are designed for different roles and have different advantages and disadvantages
- **C++** was designed to be fast (as fast as C), it pays the price of allowing bugs that hard to detect
- **Java** was designed to be vary safe (avoiding lots of bugs), but is not fast and a bit long winded
- **Python** was designed so you can rapidly write powerful programmes with a small amount of code

Computer Languages

- Different computer languages are designed for different roles and have different advantages and disadvantages
- **C++** was designed to be fast (as fast as C), it pays the price of allowing bugs that hard to detect
- **Java** was designed to be vary safe (avoiding lots of bugs), but is not fast and a bit long winded
- **Python** was designed so you can rapidly write powerful programmes with a small amount of code, **but it is not fast or safe**

Problems with C++

- Amongst a number of issues that make C++ dangerous are
 - ★ Memory management
 - ★ Writing to parts of memory that you should not
 - ★ Multiple inheritance
- However, by using existing data structures (STL) and following established programming patterns these don't have to be an issue

Problems with C++

- Amongst a number of issues that make C++ dangerous are
 - ★ Memory management
 - ★ Writing to parts of memory that you should not
 - ★ Multiple inheritance
- However, by using existing data structures (STL) and following established programming patterns these don't have to be an issue

Problems with C++

- Amongst a number of issues that make C++ dangerous are
 - ★ Memory management
 - ★ Writing to parts of memory that you should not
 - ★ **Multiple inheritance**
- However, by using existing data structures (STL) and following established programming patterns these don't have to be an issue

Problems with C++

- Amongst a number of issues that make C++ dangerous are
 - ★ Memory management
 - ★ Writing to parts of memory that you should not
 - ★ Multiple inheritance, **although you seldom need to do this**
- However, by using existing data structures (STL) and following established programming patterns these don't have to be an issue

Problems with C++

- Amongst a number of issues that make C++ dangerous are
 - ★ Memory management
 - ★ Writing to parts of memory that you should not
 - ★ Multiple inheritance, although you seldom need to do this
- However, by using existing data structures (STL) and following established programming patterns these don't have to be an issue

Problems with C++

- Amongst a number of issues that make C++ dangerous are
 - ★ Memory management
 - ★ Writing to parts of memory that you should not
 - ★ Multiple inheritance, although you seldom need to do this
- However, by using existing data structures (STL) and following established programming patterns these don't have to be an issue
 - ★ Use `vector<T>`, `set<T>`, etc. not raw arrays

Problems with C++

- Amongst a number of issues that make C++ dangerous are
 - ★ Memory management
 - ★ Writing to parts of memory that you should not
 - ★ Multiple inheritance, although you seldom need to do this
- However, by using existing data structures (STL) and following established programming patterns these don't have to be an issue
 - ★ Use `vector<T>`, `set<T>`, etc. not raw arrays
 - ★ Use standard for loops (iterators)

Memory Management

- Most programming languages have two types of memory

The Stack: is the area of memory controlled by compiler for local variables, function calls, etc.

The Heap: is area that the programmer (you) can request

Memory Management

- Most programming languages have two types of memory

The Stack: is the area of memory controlled by compiler for local variables, function calls, etc.

The Heap: is area that the programmer (you) can request

Memory Management

- Most programming languages have two types of memory

The Stack: is the area of memory controlled by compiler for local variables, function calls, etc.

The Heap: is area that the programmer (you) can request, which is nice

Memory Management

- Most programming languages have two types of memory

The Stack: is the area of memory controlled by compiler for local variables, function calls, etc.

The Heap: is area that the programmer (you) can request, which is nice

- In C++ you are given the **right** to ask for memory

```
int *storage = new int[n];
```

Memory Management

- Most programming languages have two types of memory

The Stack: is the area of memory controlled by compiler for local variables, function calls, etc.

The Heap: is area that the programmer (you) can request, which is nice

- In C++ you are given the **right** to ask for memory

```
int *storage = new int[n];
```

- You have **responsibility** to free the memory

```
delete[] storage;
```

Trouble with Memory Management

- If you don't release memory acquired with `new` using `delete` you cause a **memory leak**
- Often memory leaks are no concern, but in large programs memory leaks will rapidly exhaust the computer's memory, slowing down the code and eventually leading to the programme crashing
- To release a block of memory we can use:
`delete[] storage;`
- Now `storage` is a **dangling pointer** and must not be used as it is no longer valid
- If we accidentally delete the storage twice we get an *undefined behaviour*, but often the programme will crash

Trouble with Memory Management

- If you don't release memory acquired with `new` using `delete` you cause a **memory leak**
- Often memory leaks are no concern, but in large programs memory leaks will rapidly exhaust the computer's memory, slowing down the code and eventually leading to the programme crashing
- To release a block of memory we can use:
`delete[] storage;`
- Now `storage` is a **dangling pointer** and must not be used as it is no longer valid
- If we accidentally delete the storage twice we get an *undefined behaviour*, but often the programme will crash

Trouble with Memory Management

- If you don't release memory acquired with `new` using `delete` you cause a **memory leak**
- Often memory leaks are no concern, but in large programs memory leaks will rapidly exhaust the computer's memory, slowing down the code and eventually leading to the programme crashing
- To release a block of memory we can use:
`delete [] storage;`
- Now `storage` is a **dangling pointer** and must not be used as it is no longer valid
- If we accidentally delete the storage twice we get an *undefined behaviour*, but often the programme will crash

Trouble with Memory Management

- If you don't release memory acquired with `new` using `delete` you cause a **memory leak**
- Often memory leaks are no concern, but in large programs memory leaks will rapidly exhaust the computer's memory, slowing down the code and eventually leading to the programme crashing
- To release a block of memory we can use:
`delete[] storage;`
- Now `storage` is a **dangling pointer** and must not be used as it is no longer valid
- If we accidentally delete the storage twice we get an *undefined behaviour*, but often the programme will crash

Trouble with Memory Management

- If you don't release memory acquired with `new` using `delete` you cause a **memory leak**
- Often memory leaks are no concern, but in large programs memory leaks will rapidly exhaust the computer's memory, slowing down the code and eventually leading to the programme crashing
- To release a block of memory we can use:
`delete[] storage;`
- Now `storage` is a **dangling pointer** and must not be used as it is no longer valid
- If we accidentally delete the storage twice we get an *undefined behaviour*, but often the programme will crash

Trouble with Memory Management

- If you don't release memory acquired with `new` using `delete` you cause a **memory leak**
- Often memory leaks are no concern, but in large programs memory leaks will rapidly exhaust the computer's memory, slowing down the code and eventually leading to the programme crashing
- To release a block of memory we can use:
`delete[] storage;`
- Now `storage` is a **dangling pointer** and must not be used as it is no longer valid
- If we accidentally delete the storage twice we get an *undefined behaviour*, but often the programme will crash

Resource Acquisition is Initialisation (RAII)

- Java and Python use garbage collectors which automatically checks whether memory can be accessed and if not it is removed
- In C++ this is your responsibility
- But there is a standard **programming pattern** to elevate the problem known as **Resource Acquisition is Initialisation (RAII)**

Wrap all resources in classes. Request the resources in the constructor and release the resource in the destructor

- When the object goes out of scope (you leave a `for` loop, function call, etc.) the destructor is called and the resource is safely released

Resource Acquisition is Initialisation (RAII)

- Java and Python use garbage collectors which automatically checks whether memory can be accessed and if not it is removed
- In C++ this is your responsibility
- But there is a standard **programming pattern** to elevate the problem known as **Resource Acquisition is Initialisation (RAII)**

Wrap all resources in classes. Request the resources in the constructor and release the resource in the destructor

- When the object goes out of scope (you leave a `for` loop, function call, etc.) the destructor is called and the resource is safely released

Resource Acquisition is Initialisation (RAII)

- Java and Python use garbage collectors which automatically checks whether memory can be accessed and if not it is removed
- In C++ this is your responsibility
- But there is a standard **programming pattern** to elevate the problem known as **Resource Acquisition is Initialisation (RAII)**

Wrap all resources in classes. Request the resources in the constructor and release the resource in the destructor

- When the object goes out of scope (you leave a `for` loop, function call, etc.) the destructor is called and the resource is safely released

Resource Acquisition is Initialisation (RAII)

- Java and Python use garbage collectors which automatically checks whether memory can be accessed and if not it is removed
- In C++ this is your responsibility
- But there is a standard **programming pattern** to elevate the problem known as **Resource Acquisition is Initialisation (RAII)**

Wrap all resources in classes. Request the resources in the constructor and release the resource in the destructor

- When the object goes out of scope (you leave a `for` loop, function call, etc.) the destructor is called and the resource is safely released

Resource Acquisition is Initialisation (RAII)

- Java and Python use garbage collectors which automatically checks whether memory can be accessed and if not it is removed
- In C++ this is your responsibility
- But there is a standard **programming pattern** to elevate the problem known as **Resource Acquisition is Initialisation (RAII)**

Wrap all resources in classes. Request the resources in the constructor and release the resource in the destructor

- When the object goes out of scope (you leave a `for` loop, function call, etc.) the destructor is called and the resource is safely released

RAII

```
template <typename T>
class container {
    private:
        T* data;

    public:
        container(unsigned n) {data = new T[n];}

};
```

```
main() {

    for (int i=0; i<1000; ++i) {
        container<int> my_container(10000);
        // do something
    }
}
```

RAII

```
template <typename T>
class container {
    private:
        T* data;

    public:
        container(unsigned n) {data = new T[n];}

};
```

```
main() {

    for (int i=0; i<1000; ++i) {
        container<int> my_container(10000);
        // do something
    }
}
```


RAII

```
template <typename T>
class container {
    private:
        T* data;

    public:
        container(unsigned n) {data = new T[n];}
        ~container(unsigned n) {delete[] data;}
};

main() {

    for (int i=0; i<1000; ++i) {
        container<int> my_container(10000);
        // do something
    }
}
```

RAII

```
template <typename T>
class container {
    private:
        T* data;

    public:
        container(unsigned n) {data = new T[n];}
        ~container(unsigned n) {delete[] data;}
};
```

```
main() {

    for (int i=0; i<1000; ++i) {
        container<int> my_container(10000);
        // do something
    }
}
```

Writing over Memory

- In C and C++ the following will compile and run

```
int *array = new int[4];  
int *a = new int[2];  
double *darray = new double[4];  
array[4] = 4;
```

- However `array[4]` has not been assigned (unlike `array[0]`, `array[1]`, `array[2]` and `array[3]`)
- The memory on the heap corresponding to the address of `array[4]` might have been assigned to `a[0]` in which case you may inadvertently have set `a[0]` to 4 leading to the program not doing what you want
- It might be that you have put an `int` into `darray[0]` which will then crash the system when you read `darray[0]`

Writing over Memory

- In C and C++ the following will compile and run

```
int *array = new int[4];  
int *a = new int[2];  
double *darray = new double[4];  
array[4] = 4;
```

- However `array[4]` has not been assigned (unlike `array[0]`, `array[1]`, `array[2]` and `array[3]`)
- The memory on the heap corresponding to the address of `array[4]` might have been assigned to `a[0]` in which case you may inadvertently have set `a[0]` to 4 leading to the program not doing what you want
- It might be that you have put an `int` into `darray[0]` which will then crash the system when you read `darray[0]`

Writing over Memory

- In C and C++ the following will compile and run

```
int *array = new int[4];  
int *a = new int[2];  
double *darray = new double[4];  
array[4] = 4;
```

- However `array[4]` has not been assigned (unlike `array[0]`, `array[1]`, `array[2]` and `array[3]`)
- The memory on the heap corresponding to the address of `array[4]` might have been assigned to `a[0]` in which case you may inadvertently have set `a[0]` to 4 leading to the program not doing what you want
- It might be that you have put an `int` into `darray[0]` which will then crash the system when you read `darray[0]`

Writing over Memory

- In C and C++ the following will compile and run

```
int *array = new int[4];  
int *a = new int[2];  
double *darray = new double[4];  
array[4] = 4;
```

- However `array[4]` has not been assigned (unlike `array[0]`, `array[1]`, `array[2]` and `array[3]`)
- The memory on the heap corresponding to the address of `array[4]` might have been assigned to `a[0]` in which case you may inadvertently have set `a[0]` to 4 leading to the program not doing what you want
- It might be that you have put an `int` into `darray[0]` which will then crash the system when you read `darray[0]`

Guarding Against Mistakes

- These are really hard problems to debug because where the program goes wrong or crashes can be very far from the assignment that caused the error
- Java takes the approach that it always tests whether you are writing in valid memory
- By default C++ doesn't even for data structures—making this check slows down random access
- Checks can also make pipeline optimisations harder to make
- The onus is on the user to use the memory correctly

Guarding Against Mistakes

- These are really hard problems to debug because where the program goes wrong or crashes can be very far from the assignment that caused the error
- Java takes the approach that it always tests whether you are writing in valid memory
- By default C++ doesn't even for data structures—making this check slows down random access
- Checks can also make pipeline optimisations harder to make
- The onus is on the user to use the memory correctly

Guarding Against Mistakes

- These are really hard problems to debug because where the program goes wrong or crashes can be very far from the assignment that caused the error
- Java takes the approach that it always tests whether you are writing in valid memory
- By default C++ doesn't even for data structures—making this check slows down random access
- Checks can also make pipeline optimisations harder to make
- The onus is on the user to use the memory correctly

Guarding Against Mistakes

- These are really hard problems to debug because where the program goes wrong or crashes can be very far from the assignment that caused the error
- Java takes the approach that it always tests whether you are writing in valid memory
- By default C++ doesn't even for data structures—making this check slows down random access
- Checks can also make pipeline optimisations harder to make
- The onus is on the user to use the memory correctly

Guarding Against Mistakes

- These are really hard problems to debug because where the program goes wrong or crashes can be very far from the assignment that caused the error
- Java takes the approach that it always tests whether you are writing in valid memory
- By default C++ doesn't even for data structures—making this check slows down random access
- Checks can also make pipeline optimisations harder to make
- The onus is on the user to use the memory correctly

Guarding Against Mistakes

- These are really hard problems to debug because where the program goes wrong or crashes can be very far from the assignment that caused the error
- Java takes the approach that it always tests whether you are writing in valid memory
- By default C++ doesn't even for data structures—making this check slows down random access
- Checks can also make pipeline optimisations harder to make
- The onus is on the user to use the memory correctly

Follow Programming Idioms

- Using common data structures and following common idioms will prevent most errors

```
int n = 5;  
vector<int> array(n);
```

```
for(int i=0; i<array.size(); ++i) {  
    array[i] = i;  
}
```

```
for(auto pt=array.begin(); pt != array.end(); ++pt){  
    *pt *= 2  
}
```

```
for(int& element: array) {  
    element += 2;  
}
```

Follow Programming Idioms

- Using common data structures and following common idioms will prevent most errors

```
int n = 5;  
vector<int> array(n);
```

```
for(int i=0; i<array.size(); ++i) {  
    array[i] = i;  
}
```

```
for(auto pt=array.begin(); pt != array.end(); ++pt){  
    *pt *= 2  
}
```

```
for(int& element: array) {  
    element += 2;  
}
```

Follow Programming Idioms

- Using common data structures and following common idioms will prevent most errors

```
int n = 5;  
vector<int> array(n);
```

```
for(int i=0; i<array.size(); ++i) {  
    array[i] = i;  
}
```

```
for(auto pt=array.begin(); pt != array.end(); ++pt){  
    *pt *= 2  
}
```

```
for(int& element: array) {  
    element += 2;  
}
```

Outline

1. Why Arrays?
2. Variable Length Arrays
3. Programming Language
4. **Implementing Stacks**



Stacks

- Lets look at implementing a stack
- Remember a stack has methods
 - ★ `push (Object)`
 - ★ `pop ()`
 - ★ `top ()`
 - ★ `empty ()`

Stacks

- Lets look at implementing a stack
- Remember a stack has methods
 - ★ `push (Object)`
 - ★ `pop ()`
 - ★ `top ()`
 - ★ `empty ()`

Implementation of Stack

```
template <typename T>
class MyStack
{
private:
    std::vector<T> stack;

public:
    void push(const T& obj) {stack.push_back(obj);}

    T top() const {return stack.back();}

    T pop() {
        T tmp = stack.back();
        stack.pop_back();
        return tmp;
    }

    T empty() {return stack.size()==0;}
};
```

Implementation of Stack

```
template <typename T>
class MyStack
{
private:
    std::vector<T> stack;

public:
    void push(const T& obj) {stack.push_back(obj);}

    T top() const {return stack.back();}

    T pop() {
        T tmp = stack.back();
        stack.pop_back();
        return tmp;
    }

    T empty() {return stack.size()==0;}
};
```

Implementation of Stack

```
template <typename T>
class MyStack
{
private:
    std::vector<T> stack;

public:
    void push(const T& obj) {stack.push_back(obj);}

    T top() const {return stack.back();}

    T pop() {
        T tmp = stack.back();
        stack.pop_back();
        return tmp;
    }

    T empty() {return stack.size()==0;}
};
```

Implementation of Stack

```
template <typename T>
class MyStack
{
private:
    std::vector<T> stack;

public:
    void push(const T& obj) {stack.push_back(obj);}

    T top() const {return stack.back();}

    T pop() {
        T tmp = stack.back();
        stack.pop_back();
        return tmp;
    }

    T empty() {return stack.size()==0;}
};
```

Implementation of Stack

```
template <typename T>
class MyStack
{
private:
    std::vector<T> stack;

public:
    void push(const T& obj) {stack.push_back(obj);}

    T top() const {return stack.back();}

    T pop() {
        T tmp = stack.back();
        stack.pop_back();
        return tmp;
    }

    T empty() {return stack.size()==0;}
};
```

Notes on Implementation

- I don't need to write a constructor as C++ generates a default constructor that will initialise the stack correctly
- I don't need to write a destructor because by default the destructor for `vector<T>` will be called which releases memory
- I've written the `pop` command, that I like, but if I run

```
stack<Widget> widget_stack;  
Widget w;  
widget_stack.push(w);  
Widget w1(widget.pop());
```

if the last command throws an exception then the last term on the stack is lost for ever

Notes on Implementation

- I don't need to write a constructor as C++ generates a default constructor that will initialise the stack correctly
- I don't need to write a desctuctor because by default the destructor for `vector<T>` will be called which releases memory
- I've written the `pop` command, that I like, but if I run

```
stack<Widget> widget_stack;  
Widget w;  
widget_stack.push(w);  
Widget w1(widget.pop());
```

if the last command throws an exception then the last term on the stack is lost for ever

Notes on Implementation

- I don't need to write a constructor as C++ generates a default constructor that will initialise the stack correctly
- I don't need to write a desctuctor because by default the destructor for `vector<T>` will be called which releases memory
- I've written the `pop` command, that I like, but if I run

```
stack<Widget> widget_stack;  
Widget w;  
widget_stack.push(w);  
Widget w1(widget.pop());
```

if the last command throws an exception then the last term on the stack is lost for ever

Why not use a vector

- Surely it is mad to use `MyStack<T>` as I could just use the more powerful `vector<T>`
- I can make `MyStack<T>` as efficient as `vector<T>` by inlining function calls
- But why would I want to lose functional?
- By using `MyStack<T>` I am **declaring my intention** of using this data structure as a stack
- I'm not going to do something weird like modify an element inside the stack
- My code becomes self-explanatory

Why not use a vector

- Surely it is mad to use `MyStack<T>` as I could just use the more powerful `vector<T>`
- I can make `MyStack<T>` as efficient as `vector<T>` by inlining function calls
- But why would I want to lose functional?
- By using `MyStack<T>` I am **declaring my intention** of using this data structure as a stack
- I'm not going to do something weird like modify an element inside the stack
- My code becomes self-explanatory

Why not use a vector

- Surely it is mad to use `MyStack<T>` as I could just use the more powerful `vector<T>`
- I can make `MyStack<T>` as efficient as `vector<T>` by inlining function calls
- But why would I want to lose functional?
- By using `MyStack<T>` I am **declaring my intention** of using this data structure as a stack
- I'm not going to do something weird like modify an element inside the stack
- My code becomes self-explanatory

Why not use a vector

- Surely it is mad to use `MyStack<T>` as I could just use the more powerful `vector<T>`
- I can make `MyStack<T>` as efficient as `vector<T>` by inlining function calls
- But why would I want to lose functional?
- By using `MyStack<T>` I am **declaring my intention** of using this data structure as a stack
- I'm not going to do something weird like modify an element inside the stack
- My code becomes self-explanatory

Why not use a vector

- Surely it is mad to use `MyStack<T>` as I could just use the more powerful `vector<T>`
- I can make `MyStack<T>` as efficient as `vector<T>` by inlining function calls
- But why would I want to lose functional?
- By using `MyStack<T>` I am **declaring my intention** of using this data structure as a stack
- I'm not going to do something weird like modify an element inside the stack
- My code becomes self-explanatory

Why not use a vector

- Surely it is mad to use `MyStack<T>` as I could just use the more powerful `vector<T>`
- I can make `MyStack<T>` as efficient as `vector<T>` by inlining function calls
- But why would I want to lose functional?
- By using `MyStack<T>` I am **declaring my intention** of using this data structure as a stack
- I'm not going to do something weird like modify an element inside the stack
- My code becomes self-explanatory

Why not use a vector

- Surely it is mad to use `MyStack<T>` as I could just use the more powerful `vector<T>`
- I can make `MyStack<T>` as efficient as `vector<T>` by inlining function calls
- But why would I want to lose functional?
- By using `MyStack<T>` I am **declaring my intention** of using this data structure as a stack
- I'm not going to do something weird like modify an element inside the stack
- My code becomes self-explanatory—I don't need to write comments as it is clear what I am doing

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications

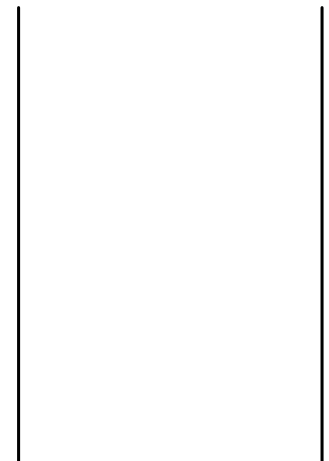
Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

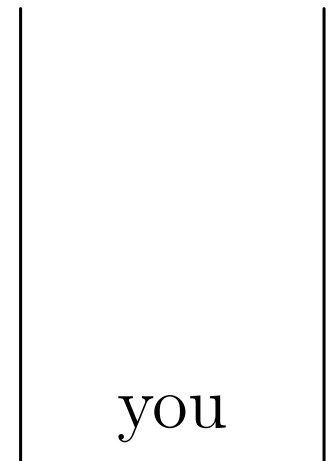
you can cage a swallow can't you



Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

can cage a swallow can't you



Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

cage a swallow can't you



can
you

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

a swallow can't you

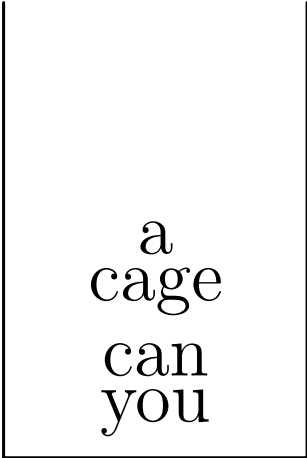


cage
can
you

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

swallow can't you



a
cage
can
you

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

can't you

swallow
a
cage
can
you

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

you

can't
swallow
a
cage
can
you

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

you
can't
swallow
a
cage
can
you

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

you

can't
swallow
a
cage
can
you

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

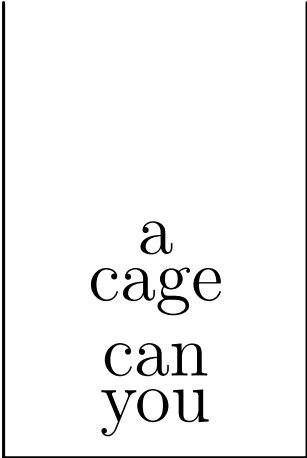
you can't

swallow
a
cage
can
you

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

you can't swallow



a
cage
can
you

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

you can't swallow a



cage
can
you

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

you can't swallow a cage

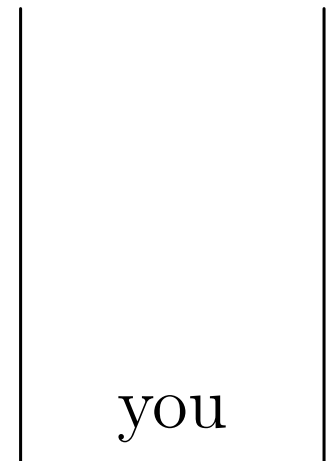


can
you

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

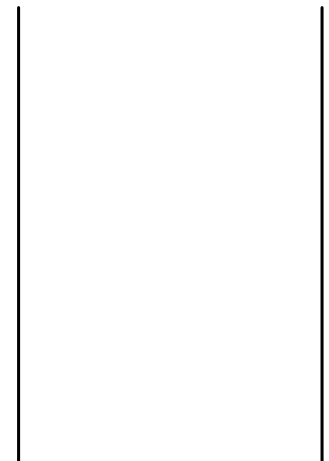
you can't swallow a cage can



Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

you can't swallow a cage can you



Reversing Strings in File

```
#include <stack>
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[]) {
    ifstream in(argv[1]);

    stack<string> stack;

    string word;
    while (in >> word)
        stack.push(word);

    while(!stack.empty()) {
        cout << stack.top() << ' ';
        stack.pop();
    }
}
```

Lessons

- Arrays are very efficient both in space (memory) and access time
- Resizing an array is not that costly
- insertion and deletion from the middle of an array are expensive, $O(n)$
- Arrays are often the simplest way to implement many other data structures, e.g. stacks
- Use (dynamically re-sizable) arrays (`vector<T>`) frequently!

Lessons

- Arrays are very efficient both in space (memory) and access time
- Resizing an array is not that costly
- insertion and deletion from the middle of an array are expensive, $O(n)$
- Arrays are often the simplest way to implement many other data structures, e.g. stacks
- Use (dynamically re-sizable) arrays (`vector<T>`) frequently!

Lessons

- Arrays are very efficient both in space (memory) and access time
- Resizing an array is not that costly
- insertion and deletion from the middle of an array are expensive, $O(n)$
- Arrays are often the simplest way to implement many other data structures, e.g. stacks
- Use (dynamically re-sizable) arrays (`vector<T>`) frequently!

Lessons

- Arrays are very efficient both in space (memory) and access time
- Resizing an array is not that costly
- insertion and deletion from the middle of an array are expensive, $O(n)$
- Arrays are often the simplest way to implement many other data structures, e.g. stacks
- Use (dynamically re-sizable) arrays (`vector<T>`) frequently!

Lessons

- Arrays are very efficient both in space (memory) and access time
- Resizing an array is not that costly
- insertion and deletion from the middle of an array are expensive, $O(n)$
- Arrays are often the simplest way to implement many other data structures, e.g. stacks
- Use (dynamically re-sizable) arrays (`vector<T>`) frequently!

Lessons

- Arrays are very efficient both in space (memory) and access time
- Resizing an array is not that costly
- insertion and deletion from the middle of an array are expensive, $O(n)$
- Arrays are often the simplest way to implement many other data structures, e.g. stacks
- Use (dynamically re-sizable) arrays (`vector<T>`) frequently!
- Stop using raw arrays