Implementing Abstractions

Pointers

- A **pointer** is a C++ variable that stores the address of an object.
- Given a pointer to an object, we can get back the original object.
  - Can then read the object's value.
  - Can then write the object's value.
- Think of a pointer as a URL for the object.

Pointers

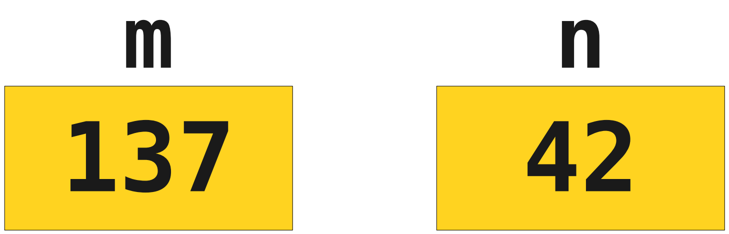
- Setting up a pointer requires two steps:
  - Declare the pointer variable.
  - Initialize the pointer variable to refer to some object.
- These are all separate steps, and forgetting any one can result in hard-to-find bugs.
- Once the pointer is set up, we can use it to read and write the object it refers to.

Pointers, Visually

```
int m = 137;
int n = 42;
```

Pointers, Visually

```
int m = 137;
int n = 42;
```



m

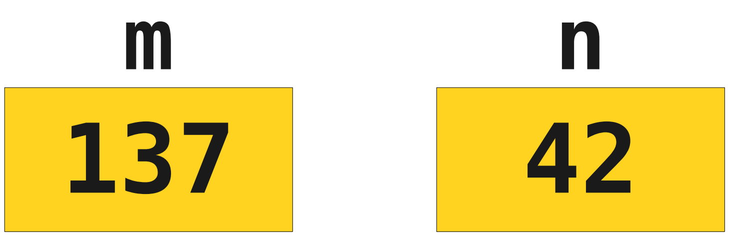
137

n

42

Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
```



m

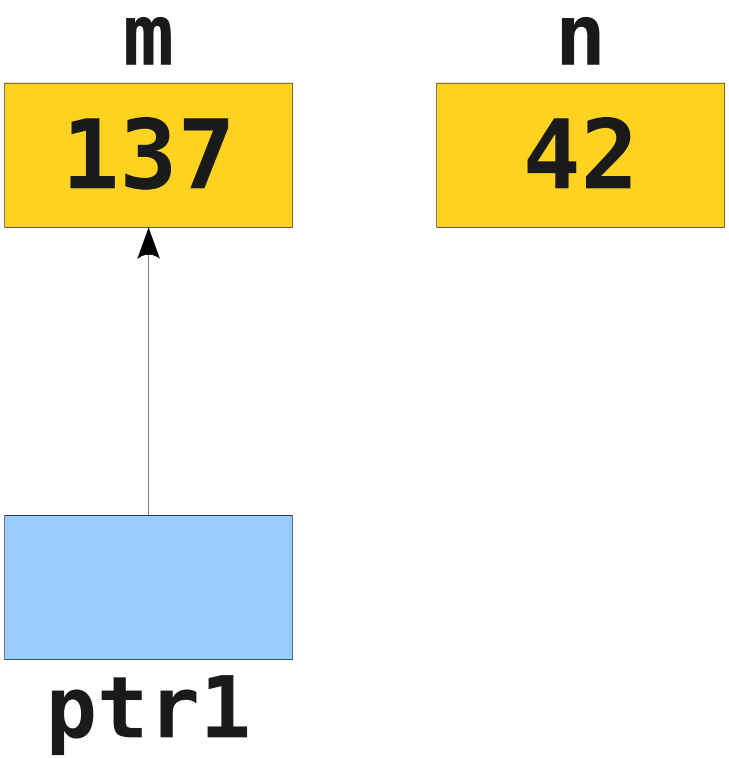
137

n

42

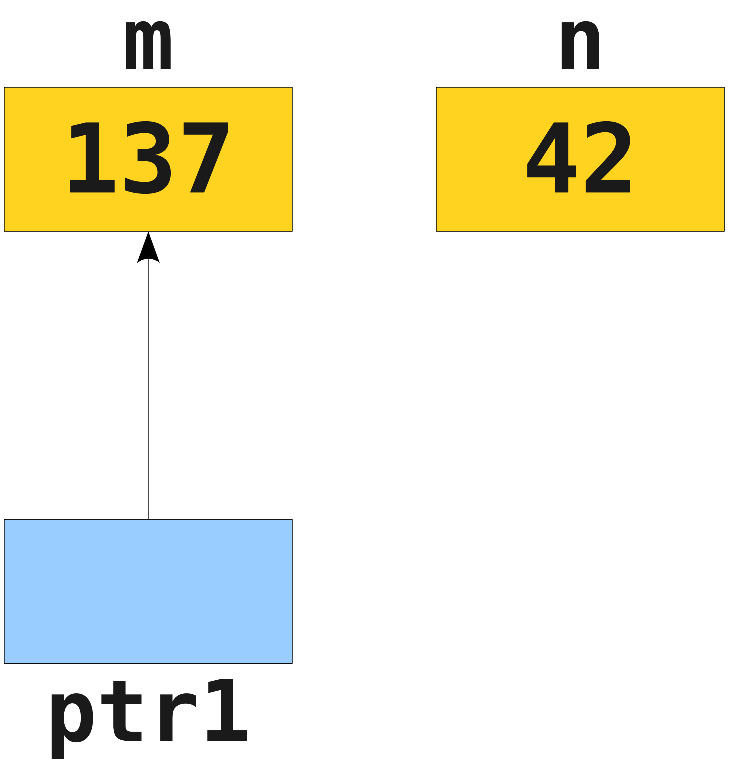
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
```



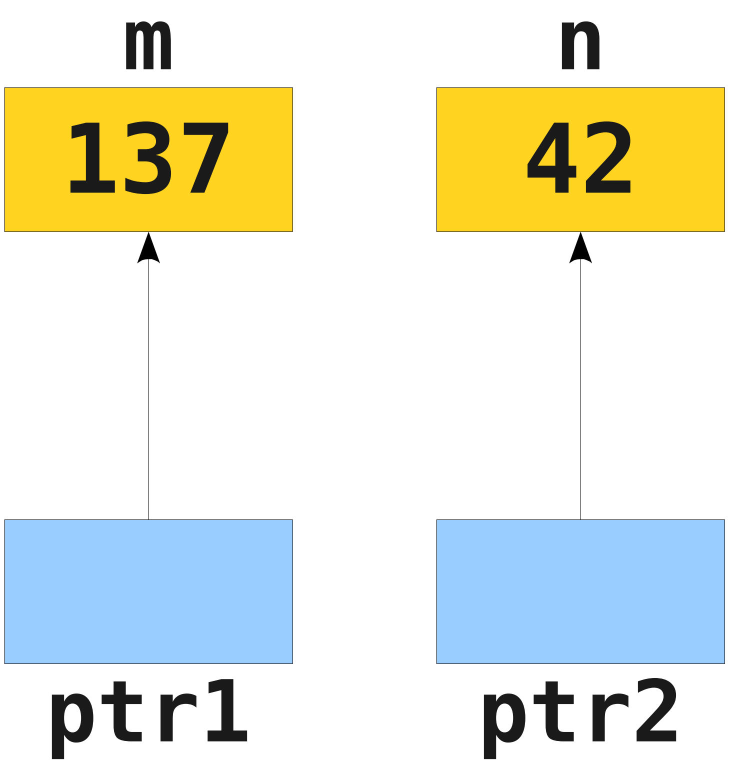
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;
```



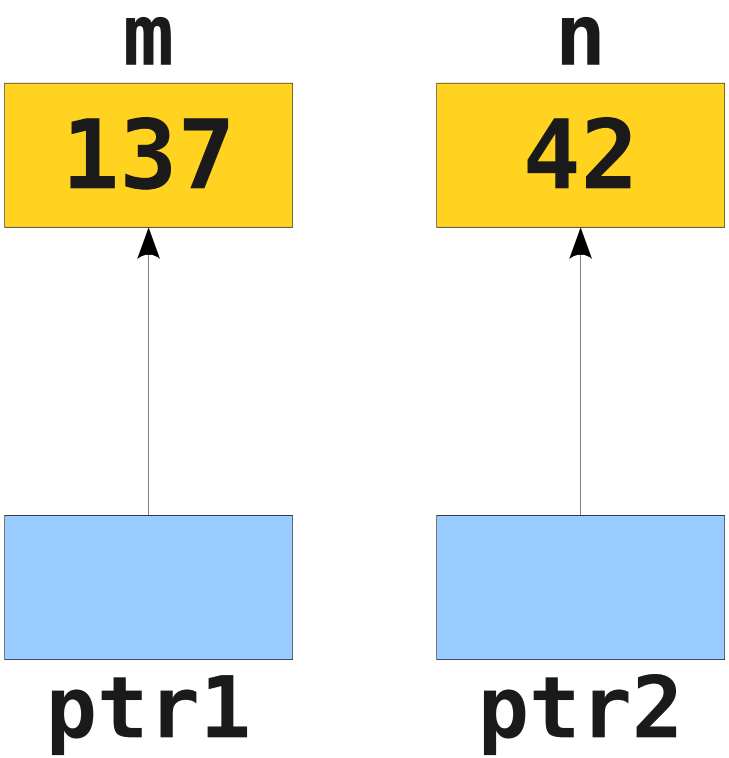
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;
```



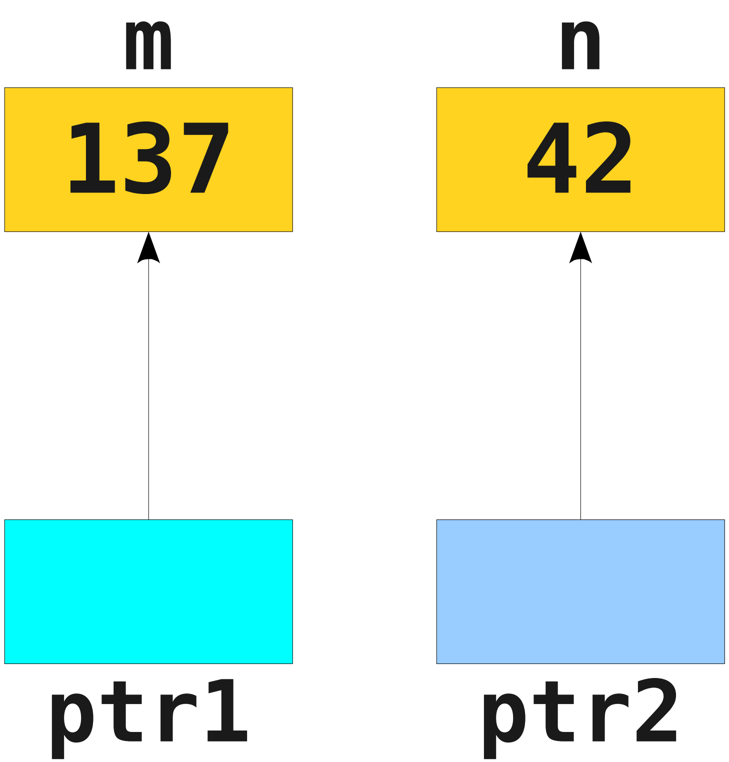
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
```



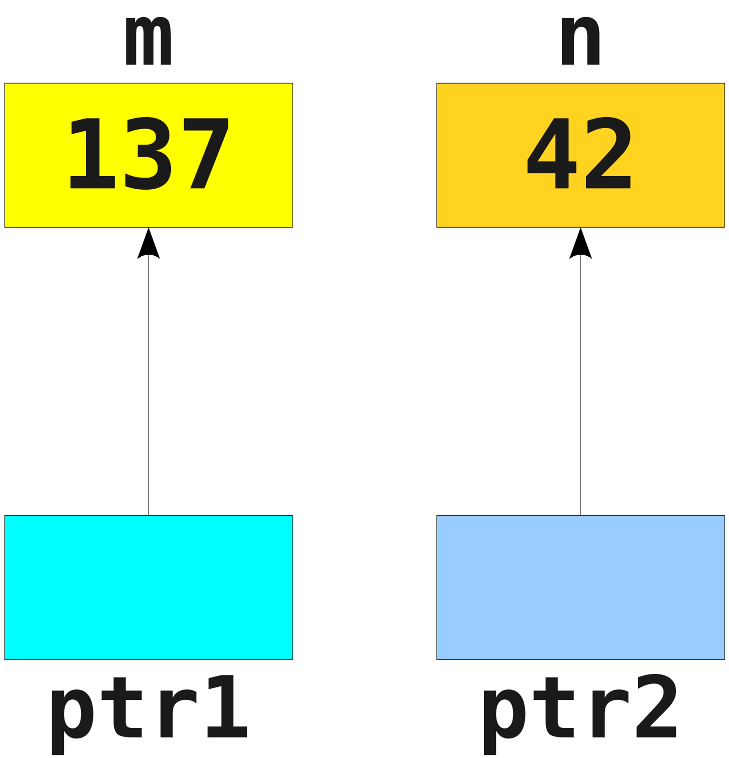
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
```



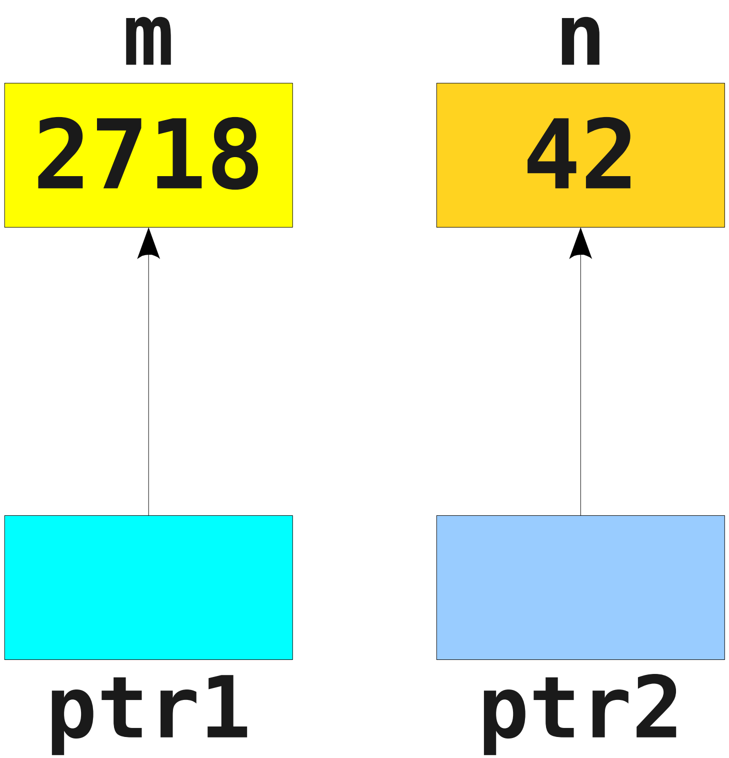
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
```



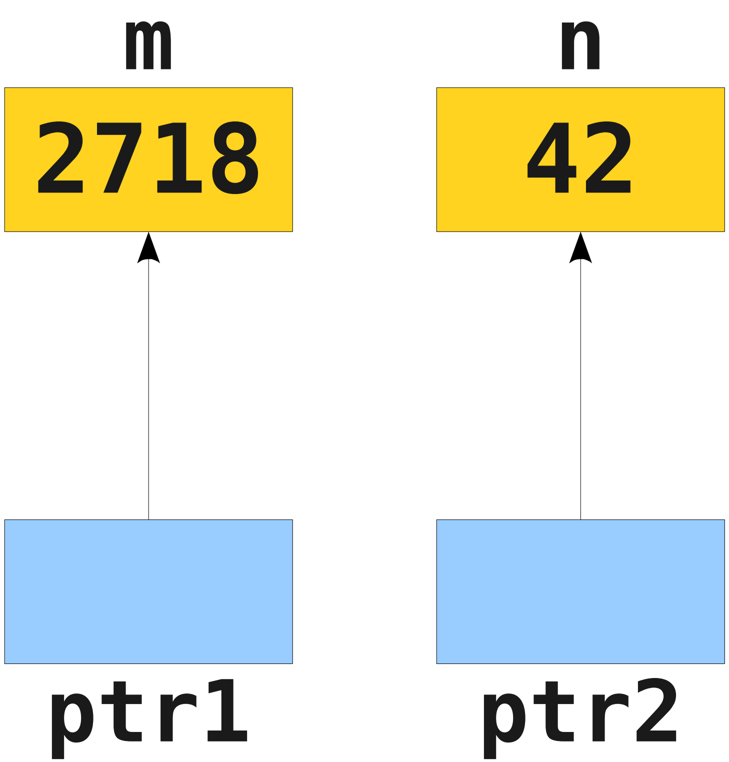
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
```



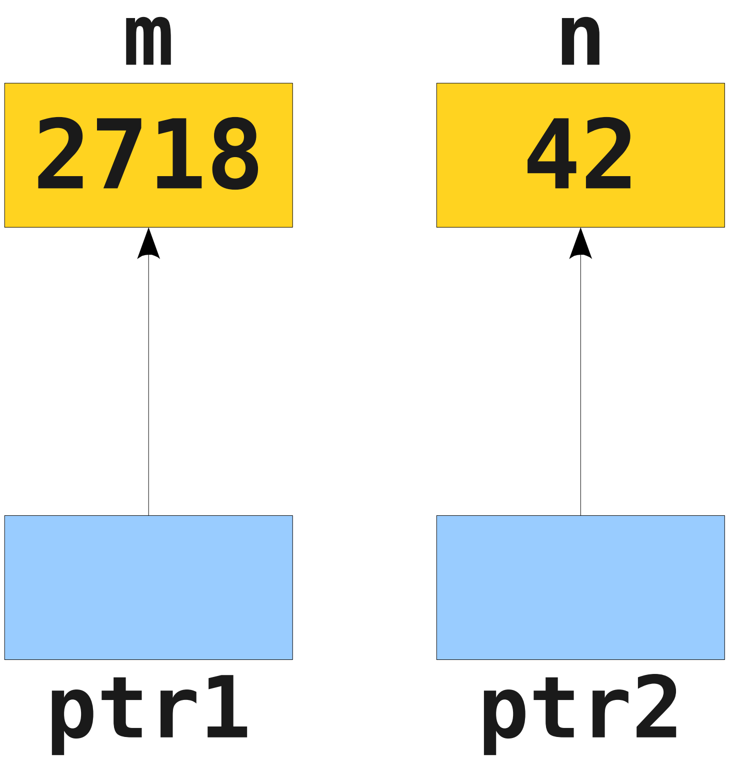
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
```



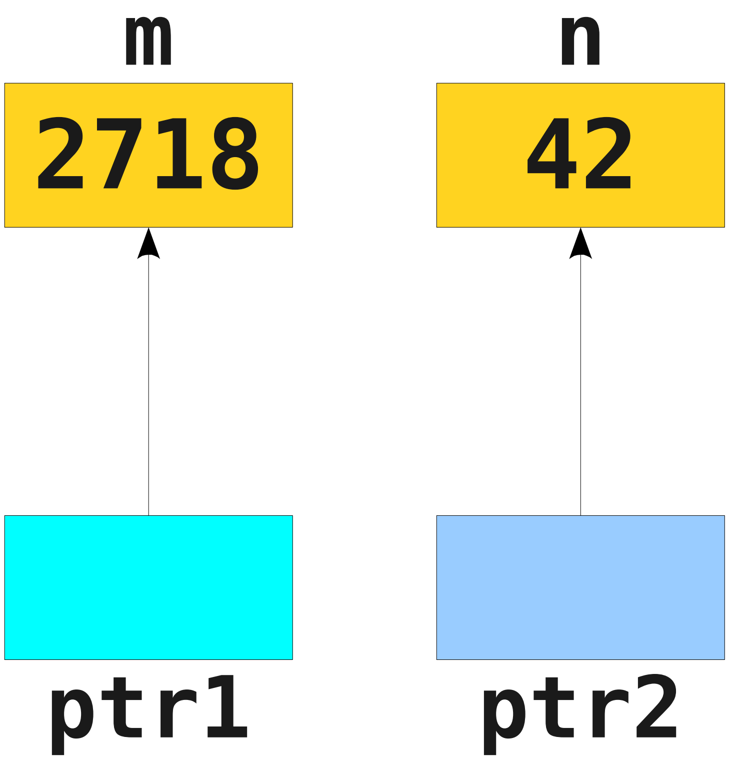
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
*ptr2 = *ptr1;
```



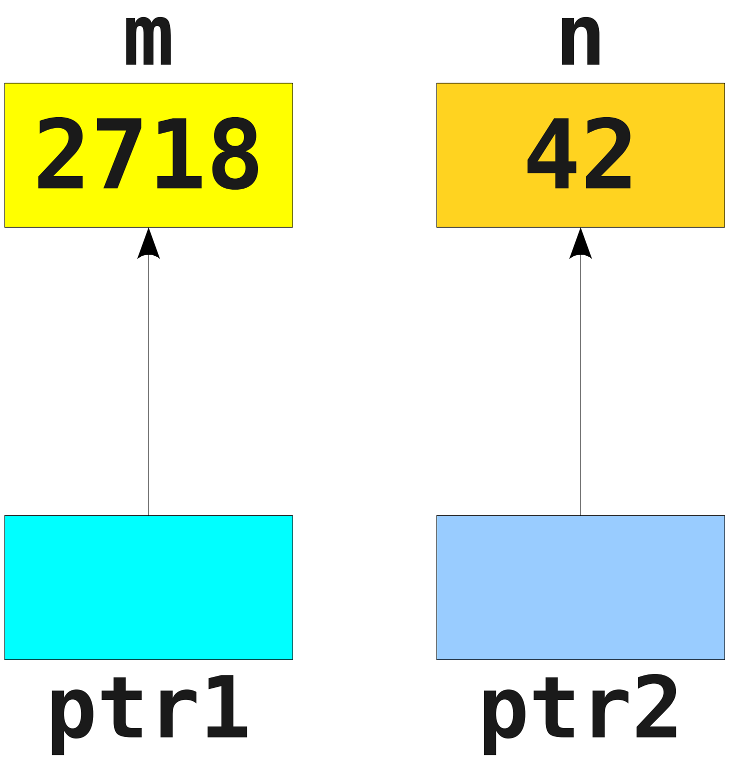
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
*ptr2 = *ptr1;
```



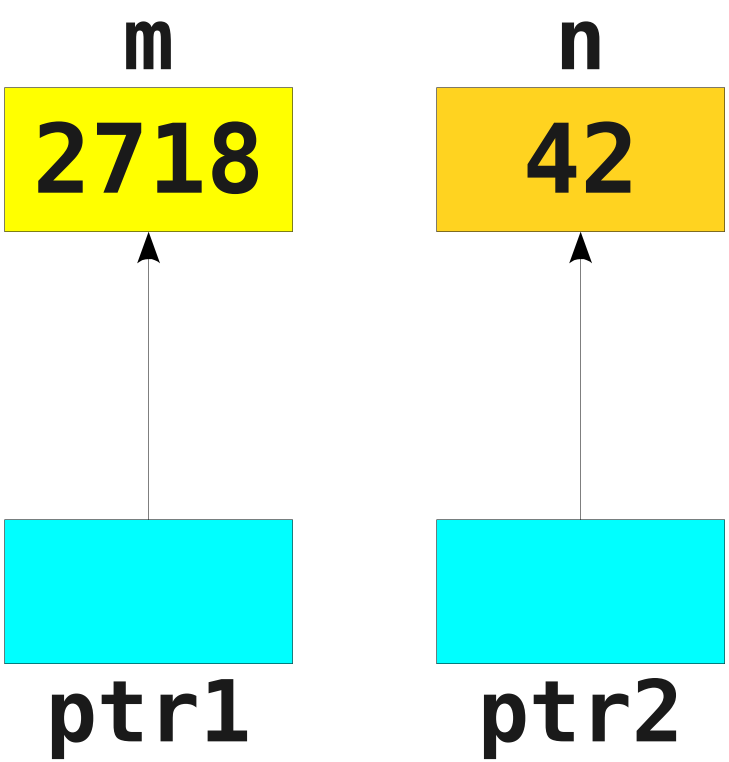
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
*ptr2 = *ptr1;
```



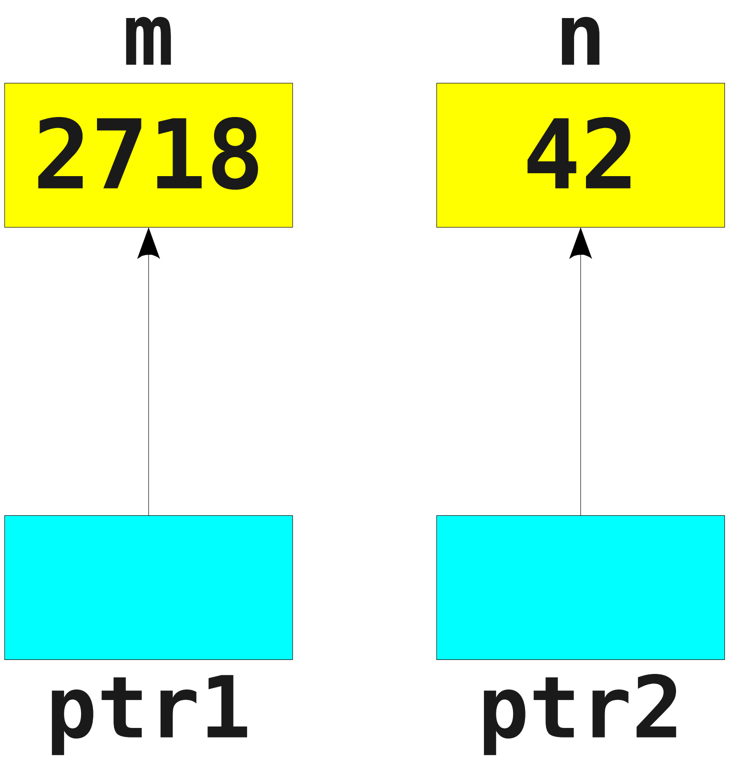
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
*ptr2 = *ptr1;
```



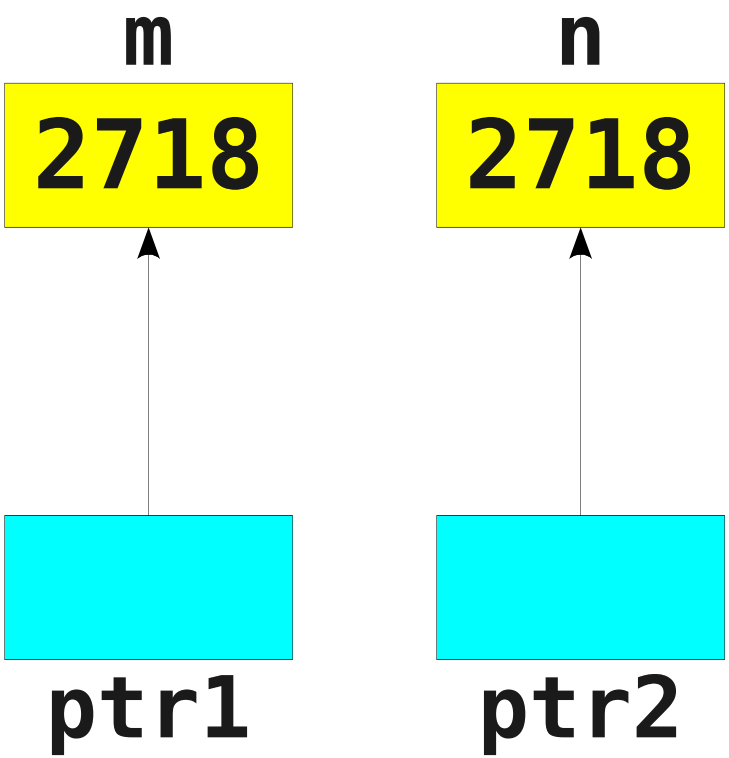
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
*ptr2 = *ptr1;
```



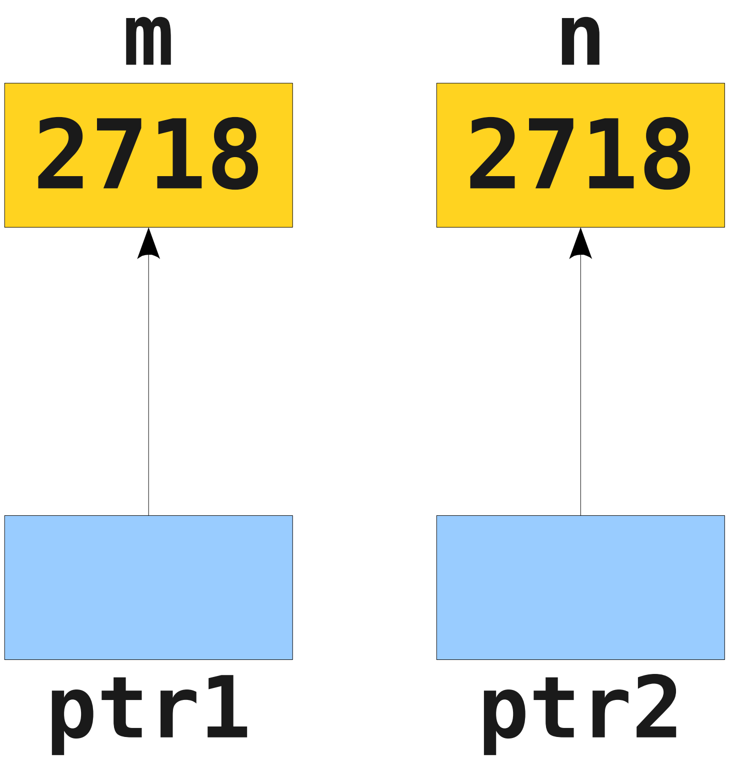
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
*ptr2 = *ptr1;
```



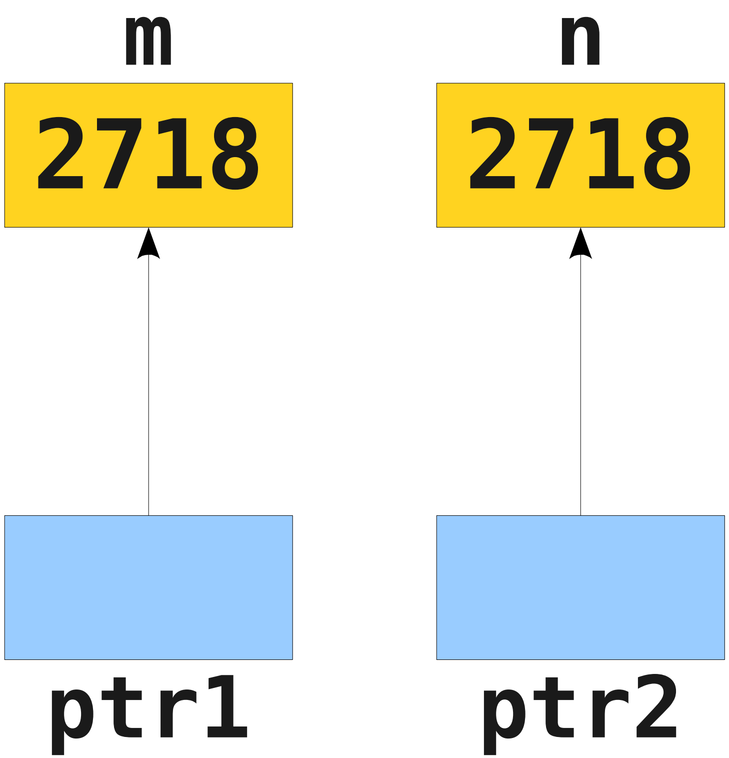
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
*ptr2 = *ptr1;
```



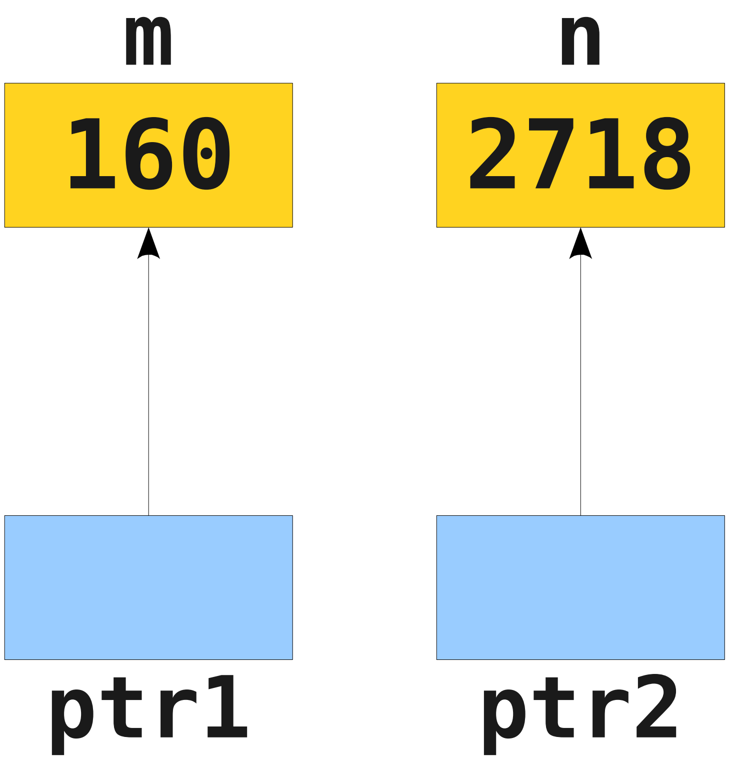
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
*ptr2 = *ptr1;
m = 160;
```



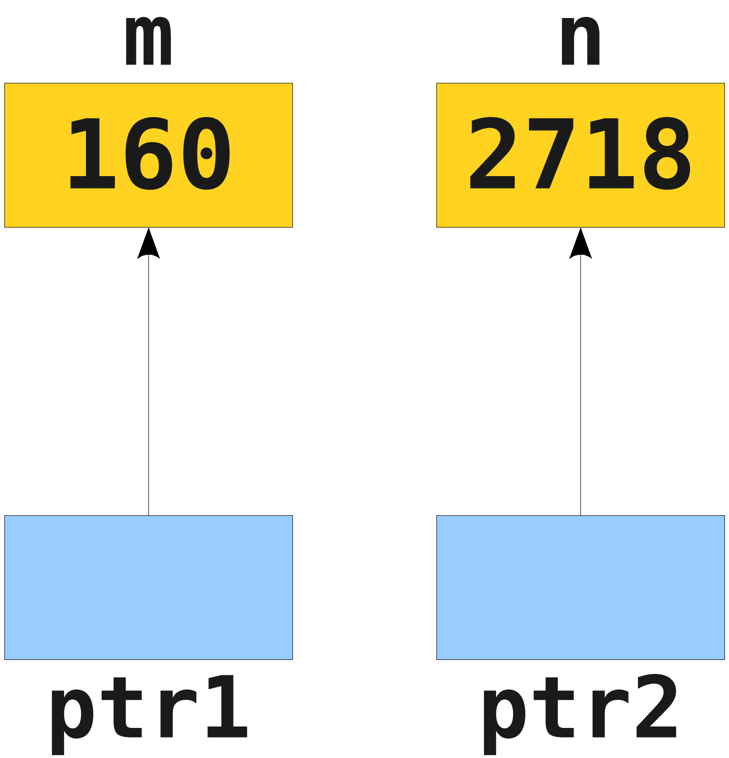
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
*ptr2 = *ptr1;
m = 160;
```



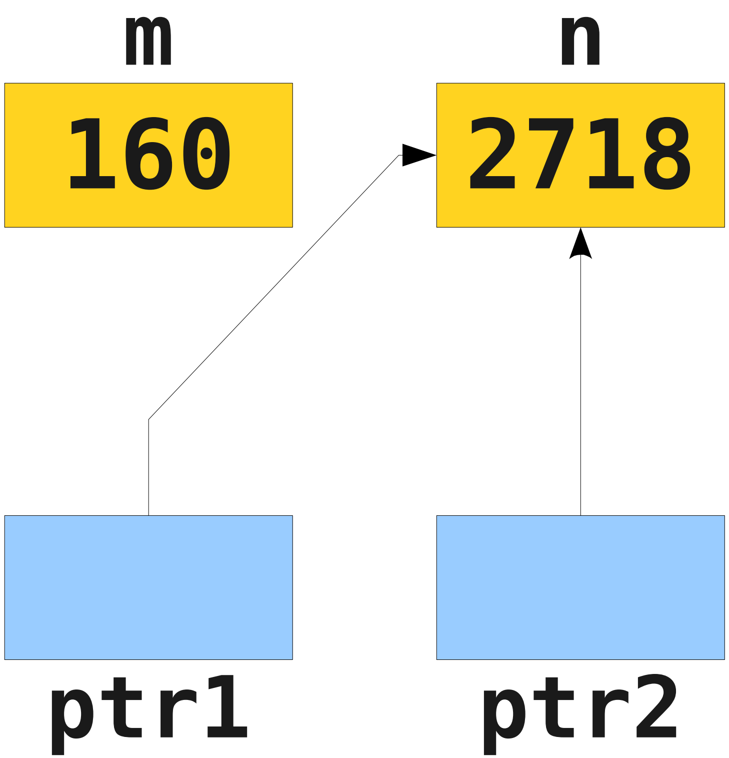
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
*ptr2 = *ptr1;
m = 160;

ptr1 = ptr2;
```



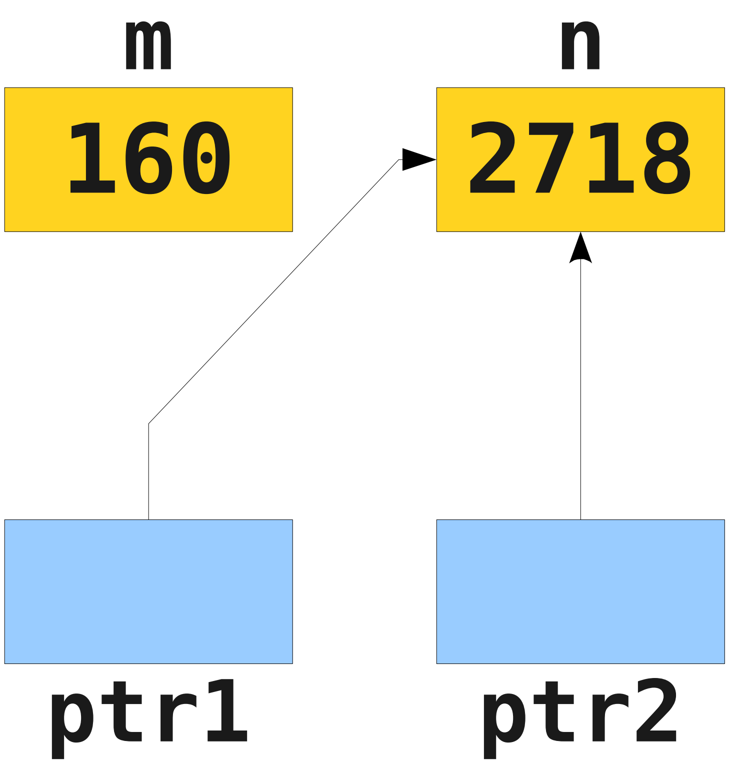
Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
*ptr2 = *ptr1;
m = 160;

ptr1 = ptr2;
```



Pointers, Visually

```
int m = 137;
int n = 42;

int* ptr1 = &m;
int* ptr2 = &n;

*ptr1 = 2718;
*ptr2 = *ptr1;
m = 160;

ptr1 = ptr2;
```



Assigning one pointer to another changes which object is being pointed at. It does not change the value of the pointee.

Why would we ever want to do this?

Allocating Multiple Objects

- One of the most important applications of pointers is **dynamic memory allocation**, the ability to construct brand-new objects at runtime.
- To allocate an array of ***n*** objects of type ***T***, use the syntax

```
new T[n]
```

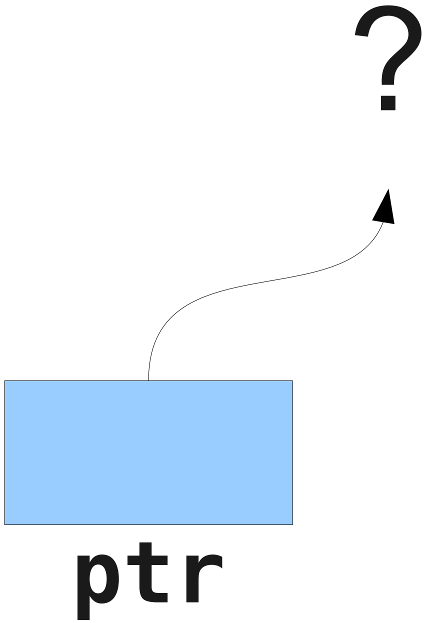
- This returns a pointer to the array of elements you have just allocated.

Dynamic Memory Allocation

```
int* ptr;
```

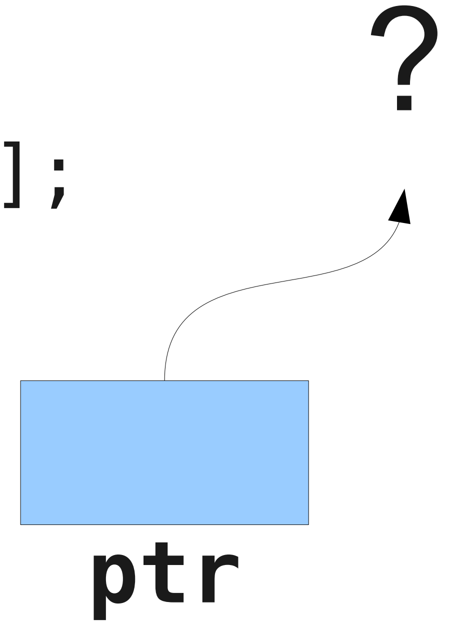
Dynamic Memory Allocation

```
int* ptr;
```



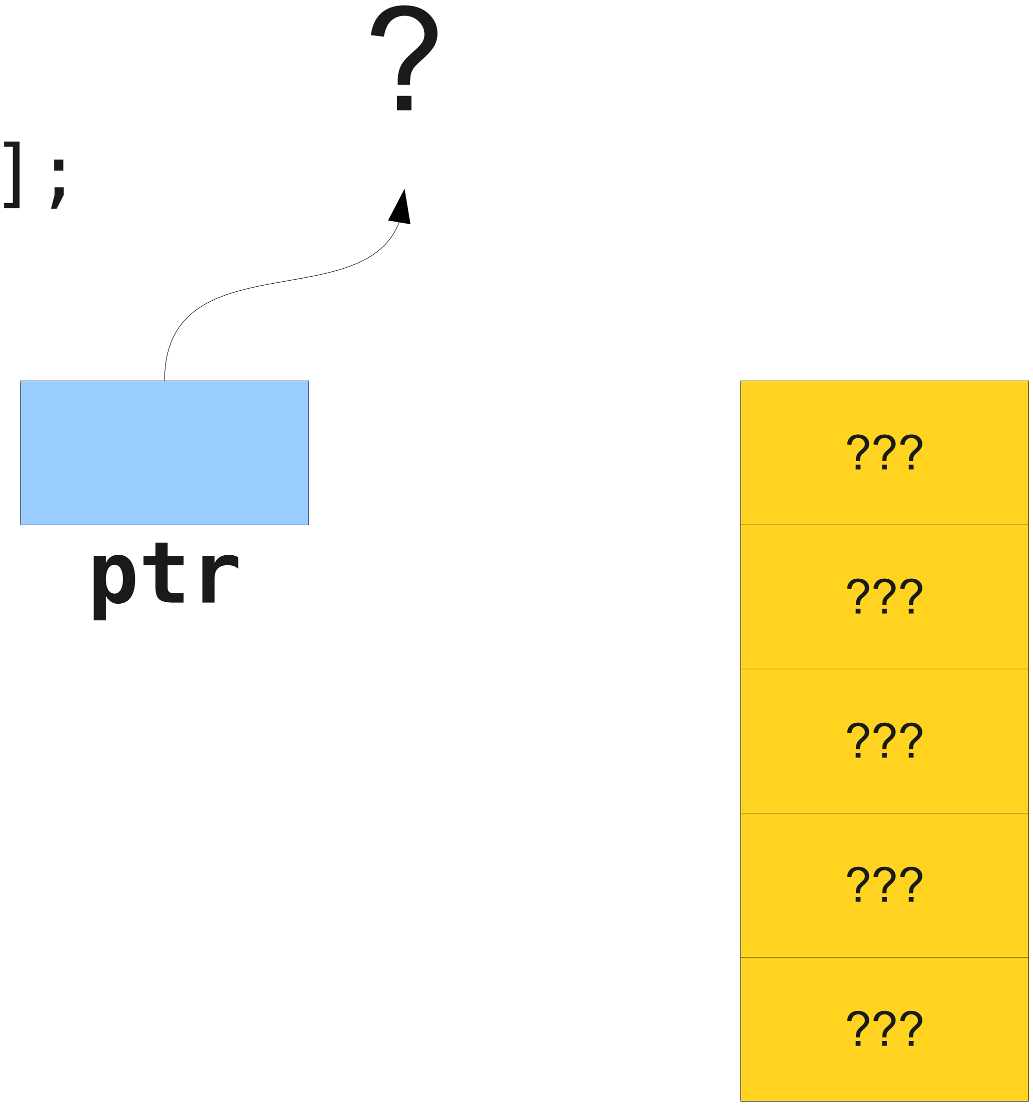
Dynamic Memory Allocation

```
int* ptr;
ptr = new int[5];
```



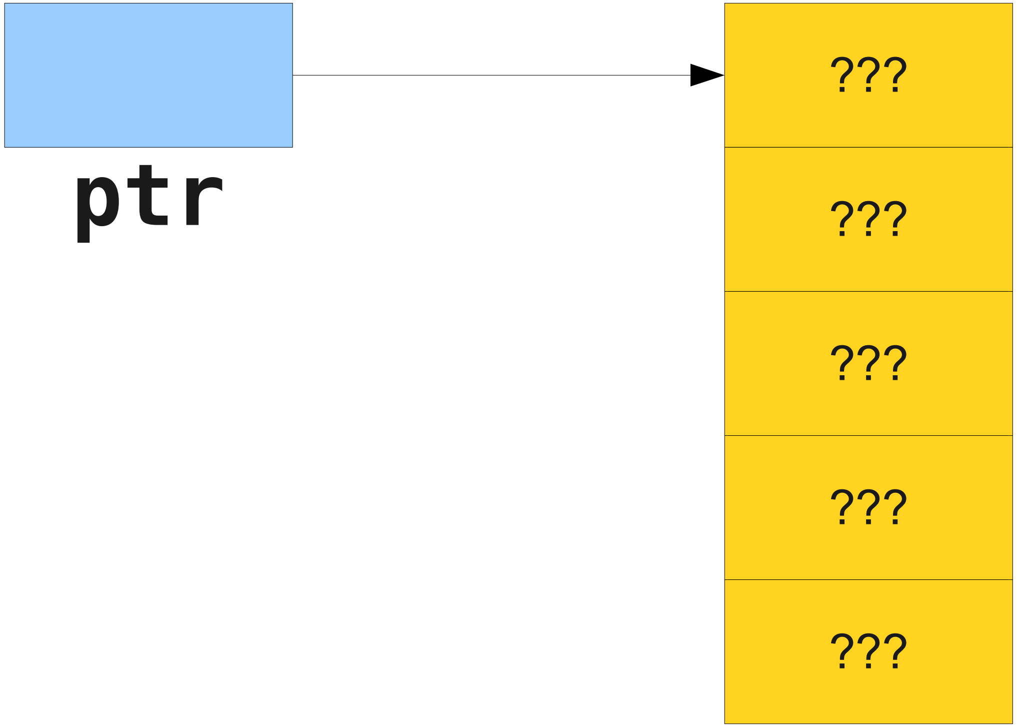
Dynamic Memory Allocation

```
int* ptr;
ptr = new int[5];
```



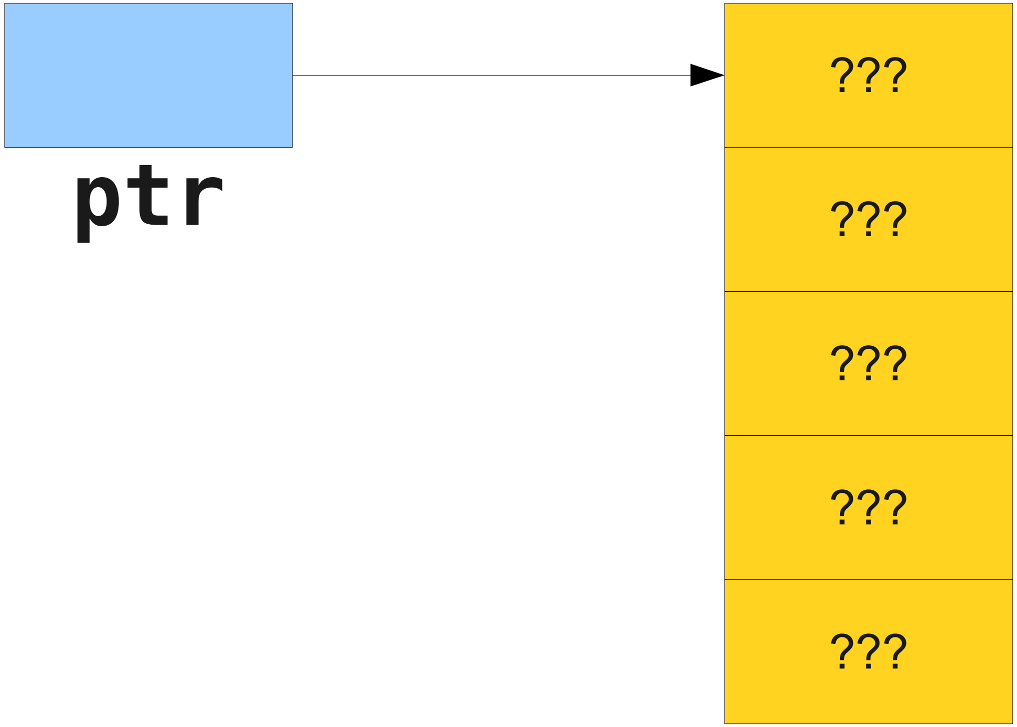
Dynamic Memory Allocation

```
int* ptr;
ptr = new int[5];
```



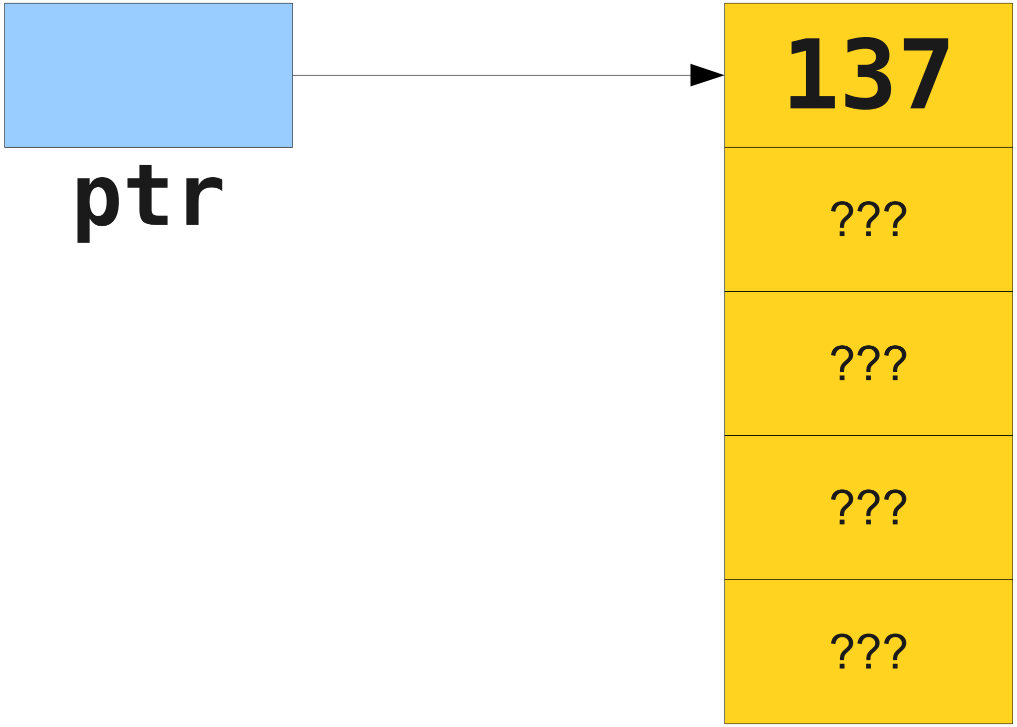
Dynamic Memory Allocation

```
int* ptr;
ptr = new int[5];

ptr[0] = 137;
```



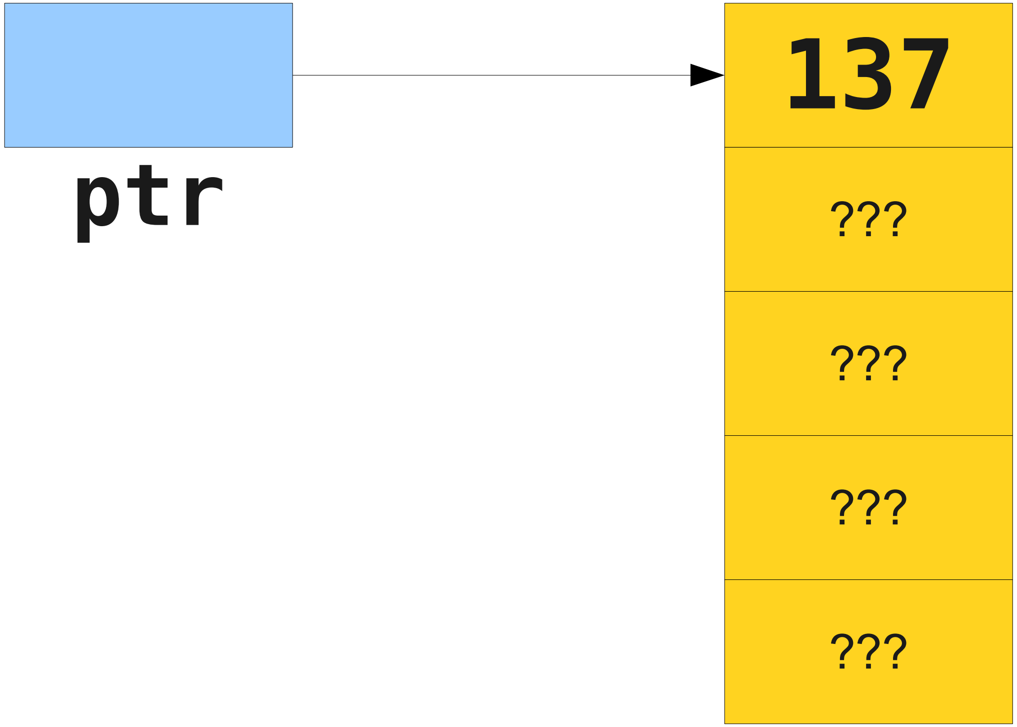
Dynamic Memory Allocation

```
int* ptr;
ptr = new int[5];

ptr[0] = 137;
```



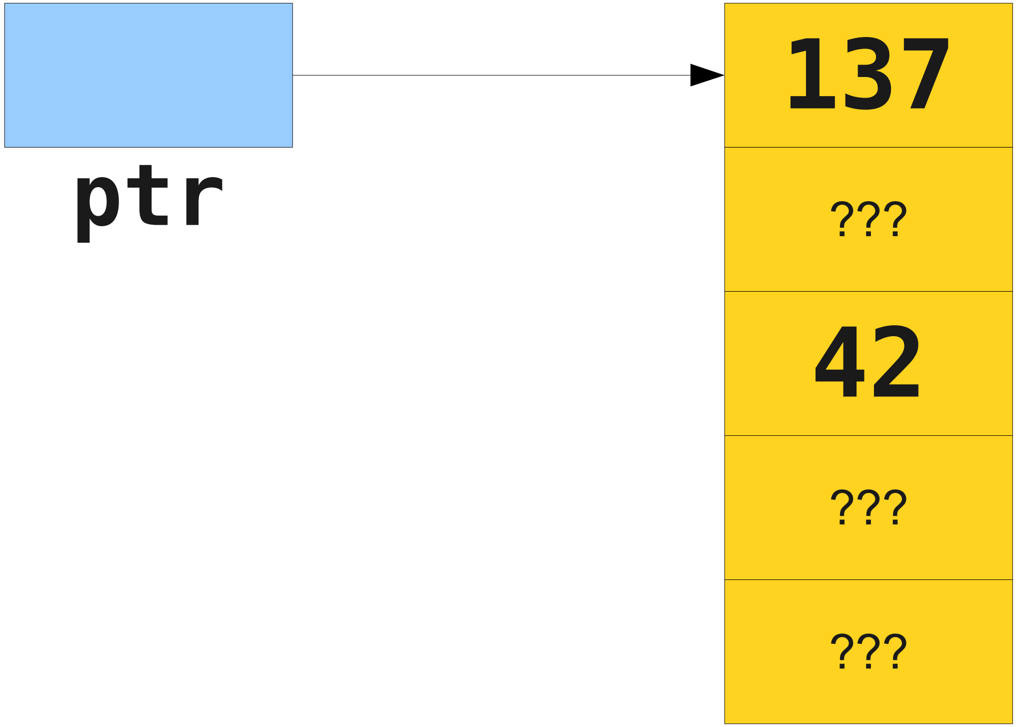
Dynamic Memory Allocation

```
int* ptr;
ptr = new int[5];

ptr[0] = 137;
ptr[2] = 42;
```



Dynamic Memory Allocation

```
int* ptr;
ptr = new int[5];

ptr[0] = 137;
ptr[2] = 42;
```



Notes on Dynamic Arrays

- Arrays in C++ do **not** know their own size.
  - You must store this separately.
- Arrays in C++ do **not** have bounds-checking.
  - You must make sure not to read off the end of the array.
- Arrays in C++ **cannot** be resized.

Cleaning Up

- Unlike other languages like Java, in C++, you are responsible for deallocating any memory allocated with **`new[]`**.
- You can deallocate memory with the **`delete[]`** operator:

```
delete[] ptr;
```

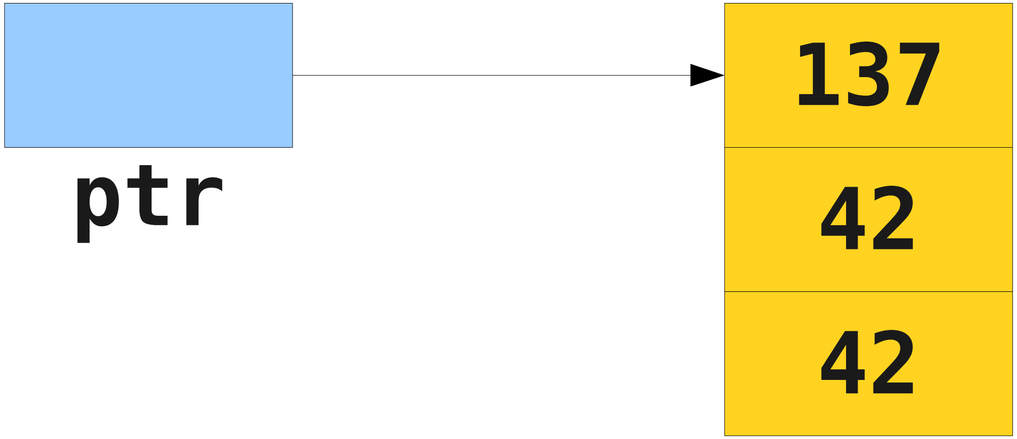
- This destroys the array pointed at by the given pointer, not the pointer itself.

Cleaning Up

- Unlike other languages like Java, in C++, you are responsible for deallocating any memory allocated with **`new[]`**.
- You can deallocate memory with the **`delete[]`** operator:

```
delete[] ptr;
```

- This destroys the array pointed at by the given pointer, not the pointer itself.

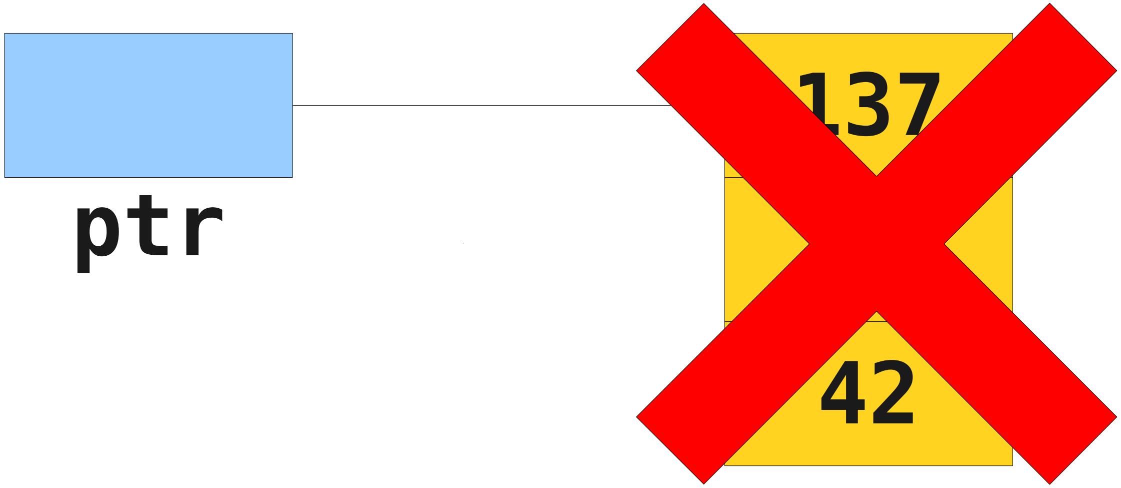


Cleaning Up

- Unlike other languages like Java, in C++, you are responsible for deallocating any memory allocated with **new[]**.
- You can deallocate memory with the **delete[]** operator:

```
delete[] ptr;
```

- This destroys the array pointed at by the given pointer, not the pointer itself.



Cleaning Up

- Unlike other languages like Java, in C++, you are responsible for deallocating any memory allocated with **`new[]`**.
- You can deallocate memory with the **`delete[]`** operator:

```
delete[] ptr;
```

- This destroys the array pointed at by the given pointer, not the pointer itself.

Words of Caution

- C++ has few of the safety features present in Java.
- All of the following result in **undefined behavior** in C++:
  - Reading or writing through a pointer that you haven't initialized.
  - Reading or writing through a pointer to memory that you have deallocated.
  - Reading off the end of an array.
  - Treating a non-array like an array.

Implementing **`Stack`**

Implementing **Stack**

- Last time, we saw how to implement **RandomBag** in terms of **Vector**.
- We could also implement **Stack** in terms of **Vector**.
- What if we wanted to implement the **Stack** without relying on any other collections?
- Let's build the stack directly!

Storing Values

- Right now, if we need to store multiple values, we can
  - Declare a whole bunch of variables,
  - Use a collections class, or
  - Dynamically allocate space.

Storing Values

- Right now, if we need to store multiple values, we can
  - ~~Declare a whole bunch of variables,~~
  - Use a collections class, or
  - Dynamically allocate space.

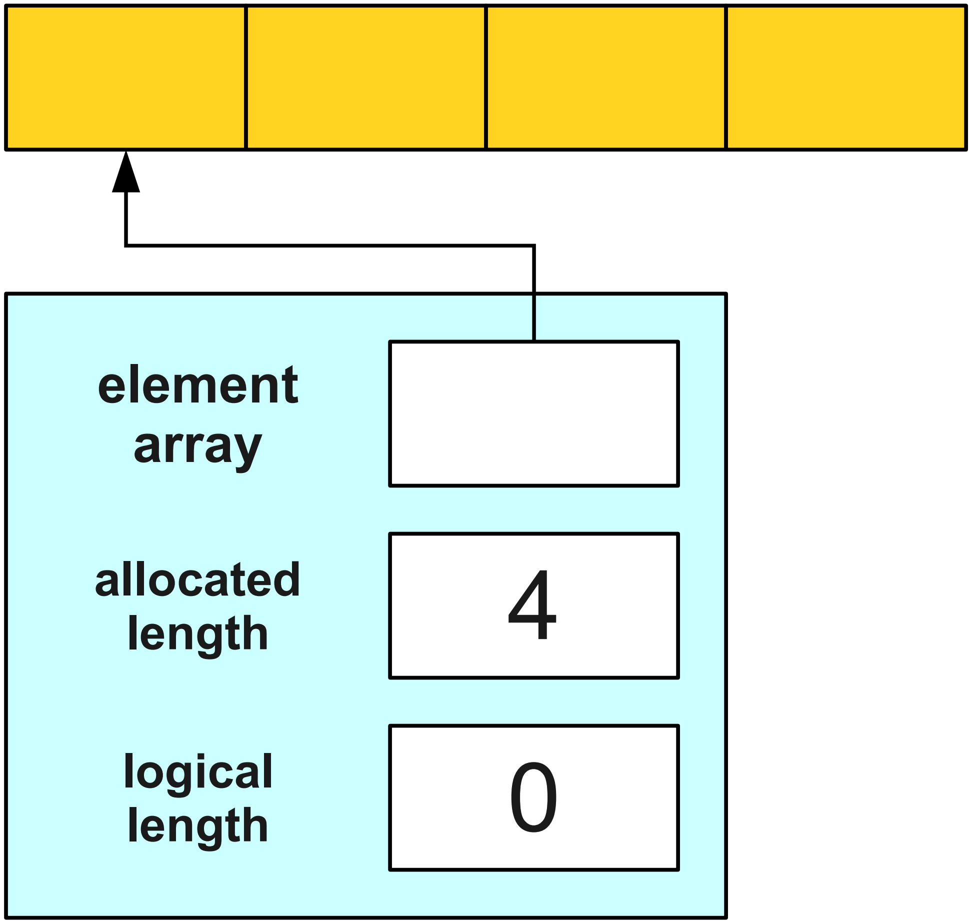
Storing Values

- Right now, if we need to store multiple values, we can
  - ~~Declare a whole bunch of variables,~~
  - ~~Use a collections class, or~~
  - Dynamically allocate space.

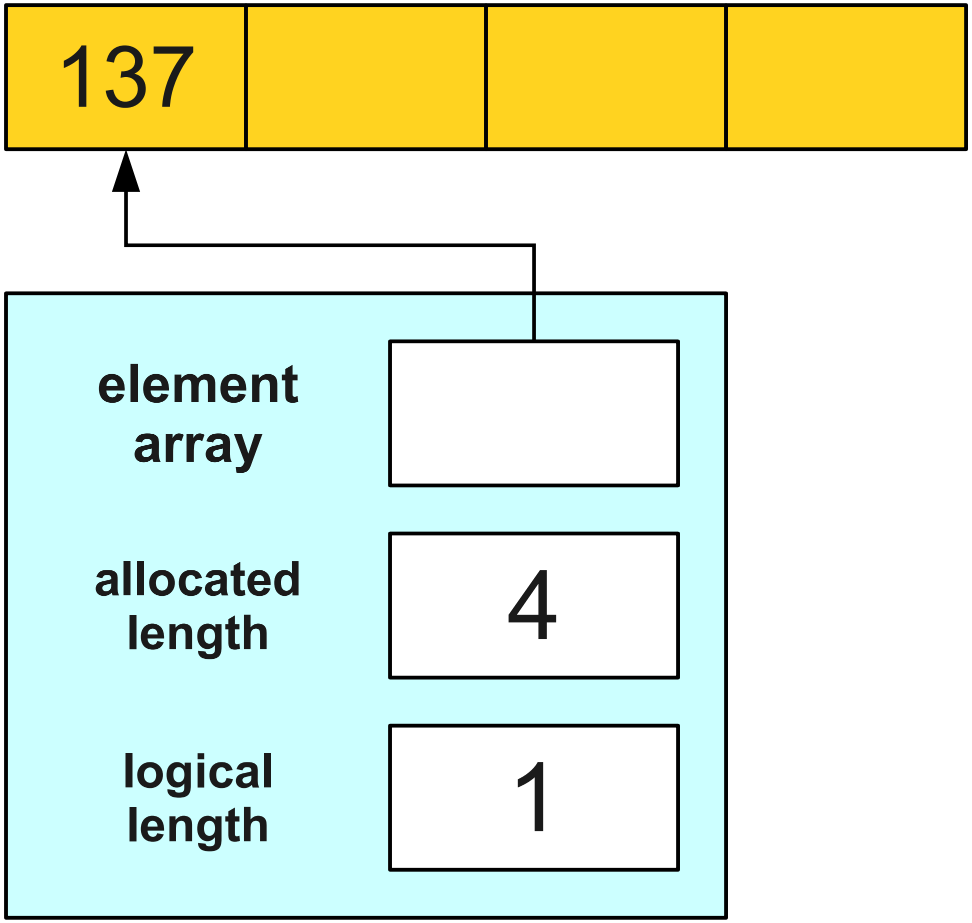
An Initial Idea

- **A bounded stack.**
- Allocate a fixed-size array for elements.
- Add elements to the array when they're pushed.
- Remove elements from the array when they're popped.
- Report an error if we exceed the size of the array.

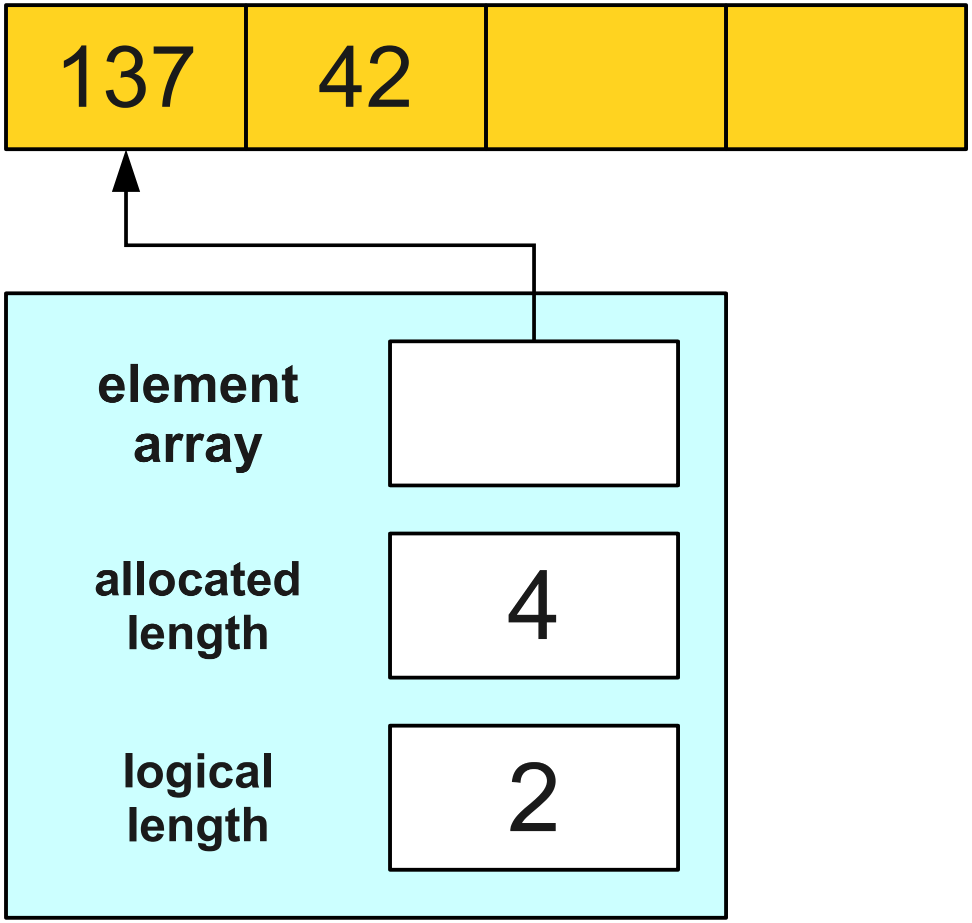
An Initial Idea



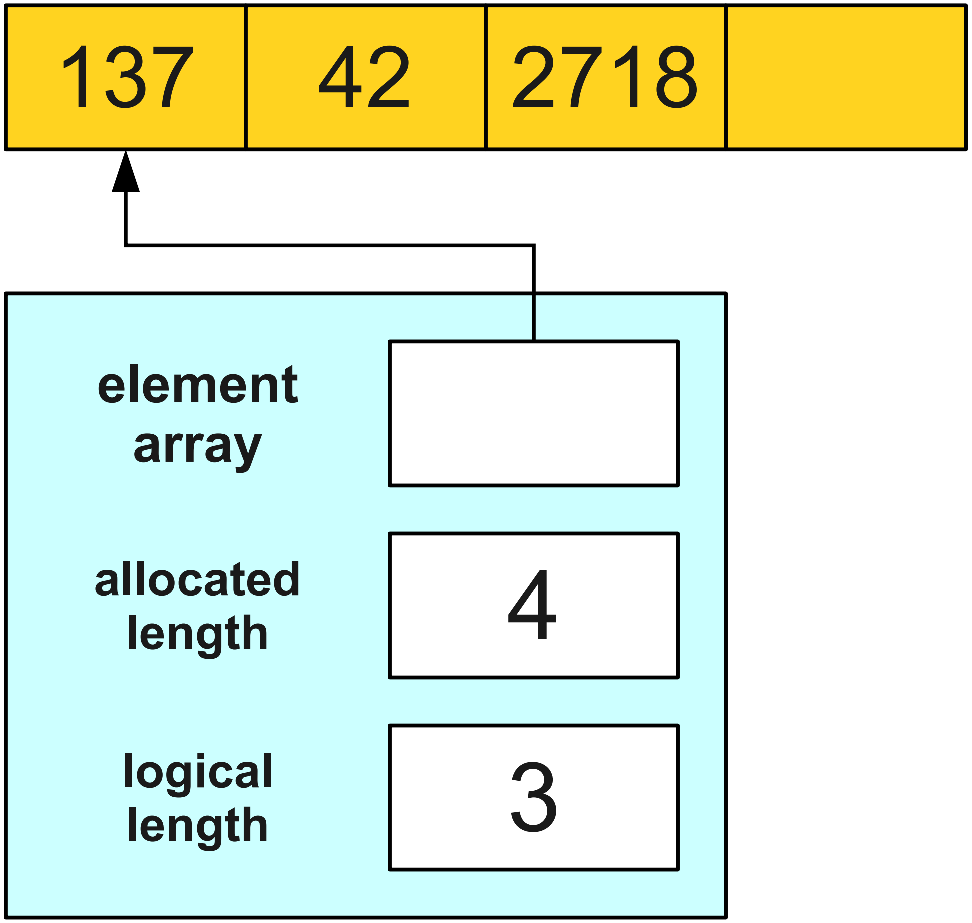
An Initial Idea



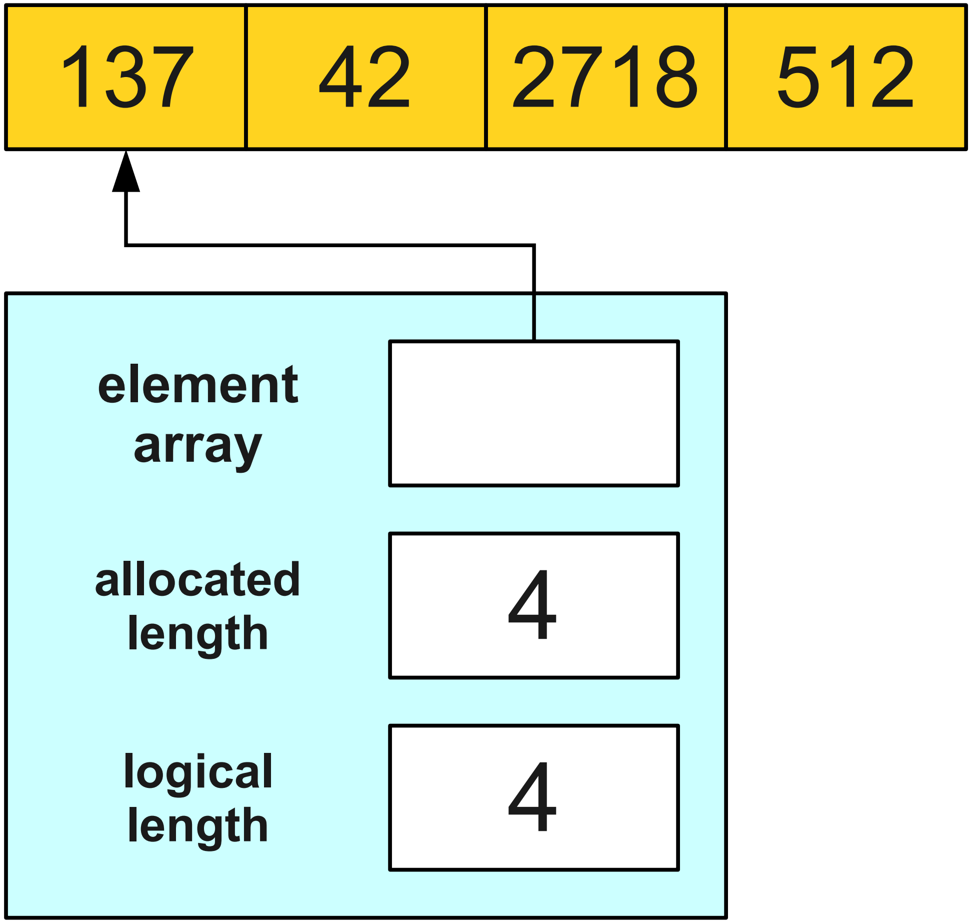
An Initial Idea



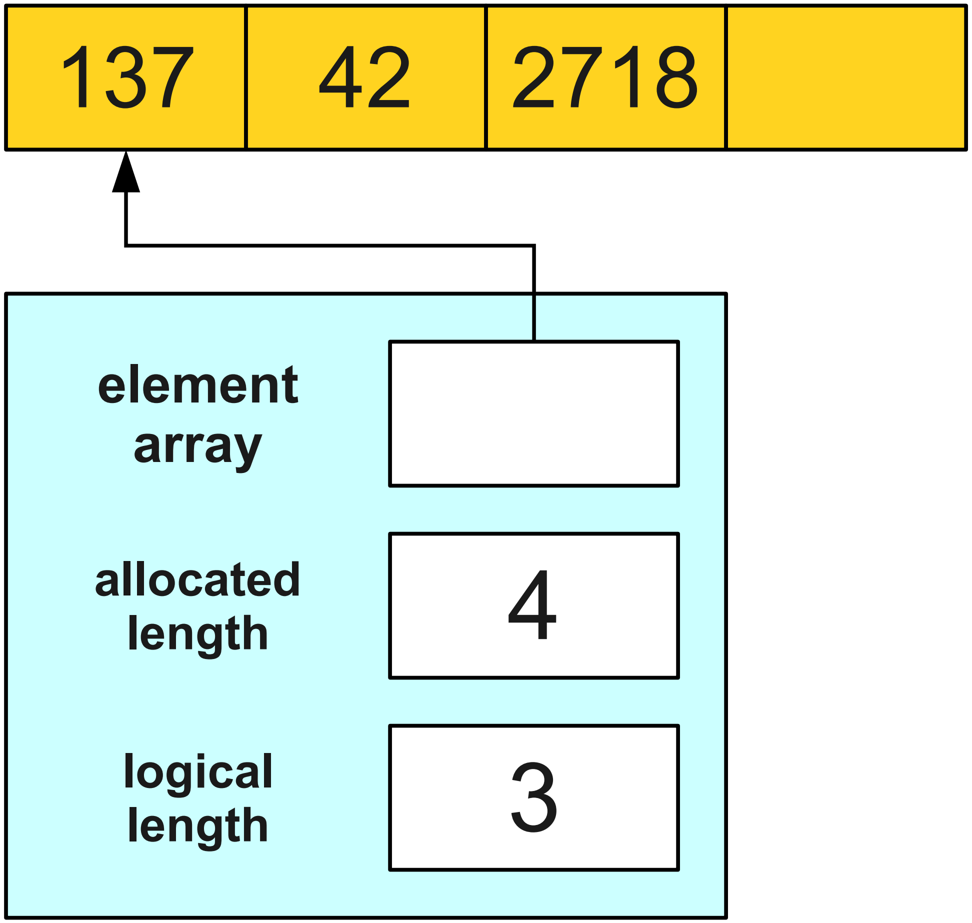
An Initial Idea



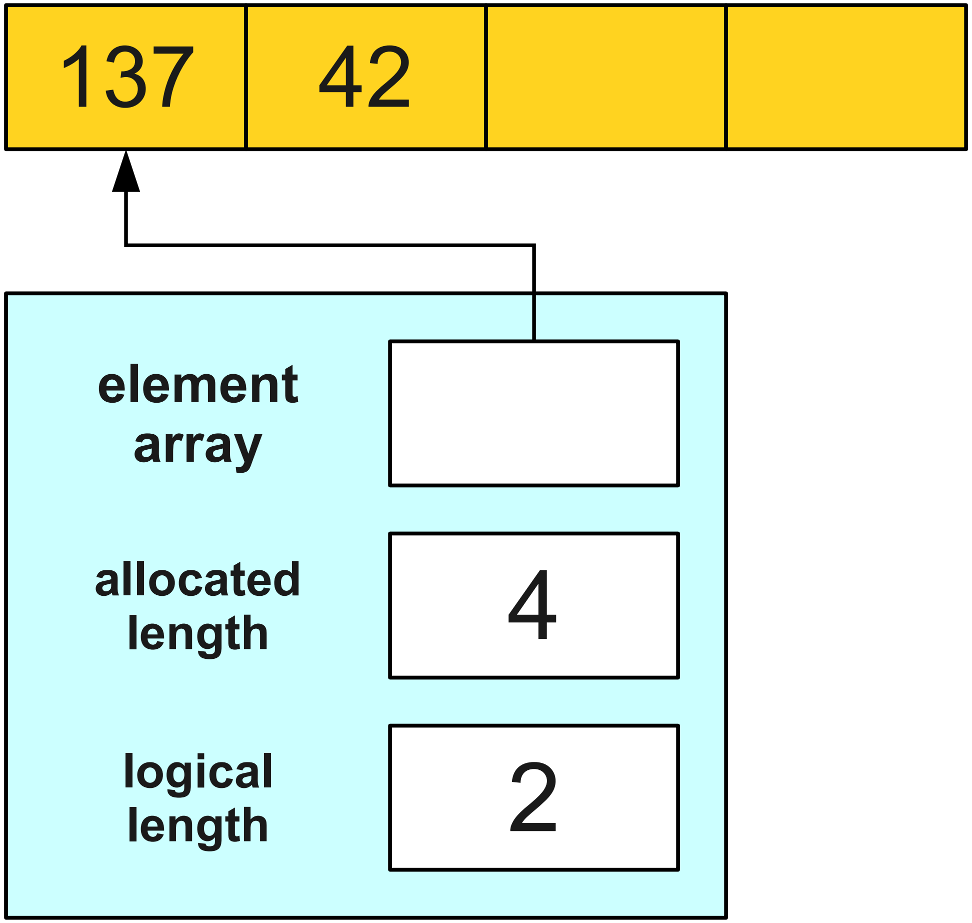
An Initial Idea



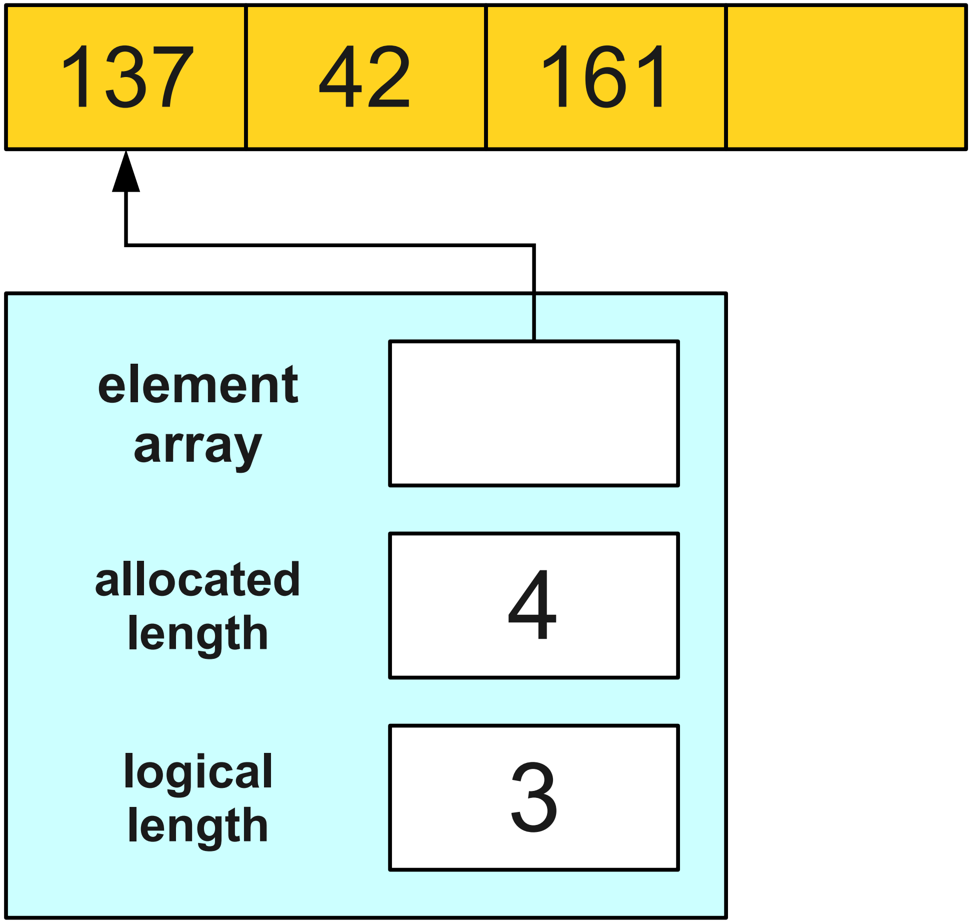
An Initial Idea



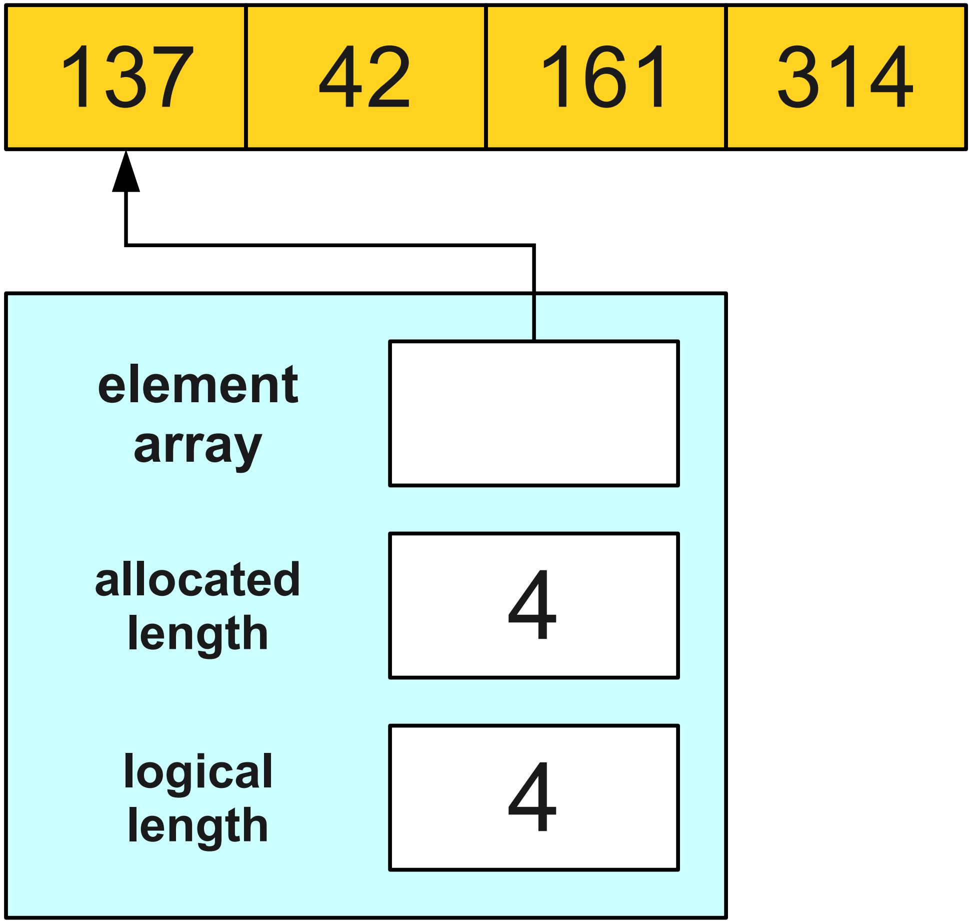
An Initial Idea



An Initial Idea



An Initial Idea



137	42	161	314

element array

allocated length: 4

logical length: 4

Let's Code it Up!

Constructors

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

```
ClassName(args);
```

Destructors

- A **destructor** is a special member function responsible for cleaning up an object's memory.
- Automatically called when a local variable goes out of scope.
- Automatically called if you `delete` a pointer to an object.
- Syntax: The constructor for a class named ***ClassName*** has signature

```
~ClassName();
```