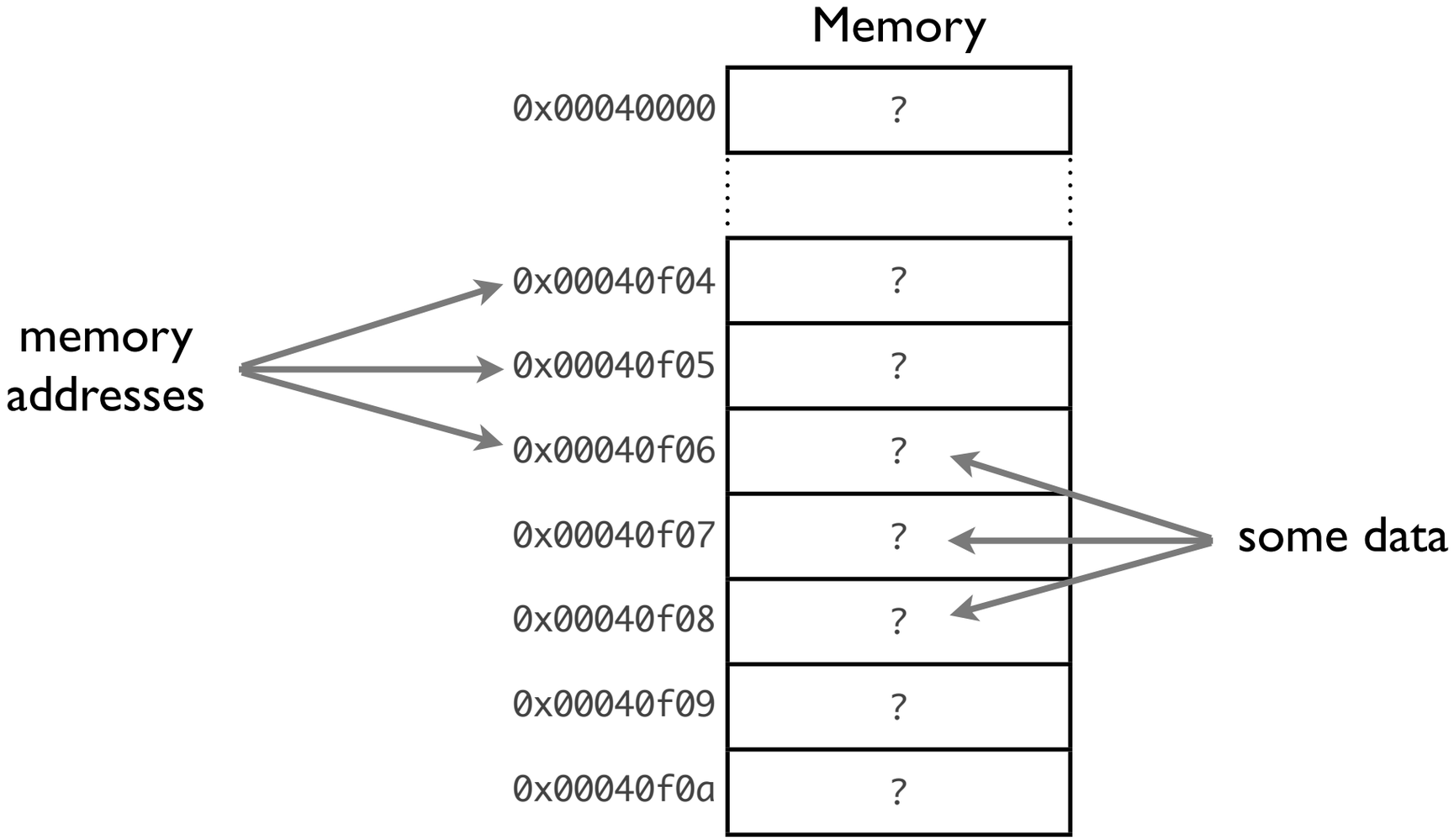
Dynamic Arrays

A Walk Down Memory Lane

Computers manipulate data stored in *memory*

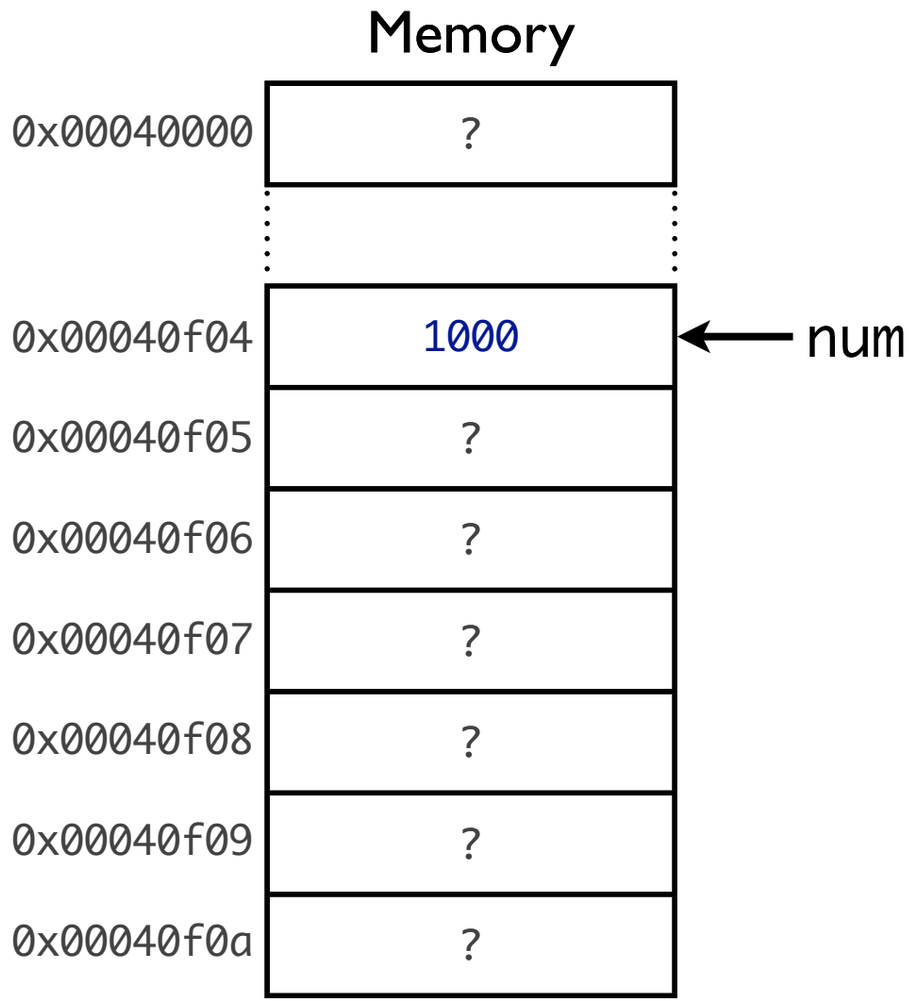
- memory is divided into byte-sized pieces, each of which can store a single value
- each 'slot' has its own unique address that is used to reference it



A Walk Down Memory Lane

Variables provide an easy way to reference memory locations

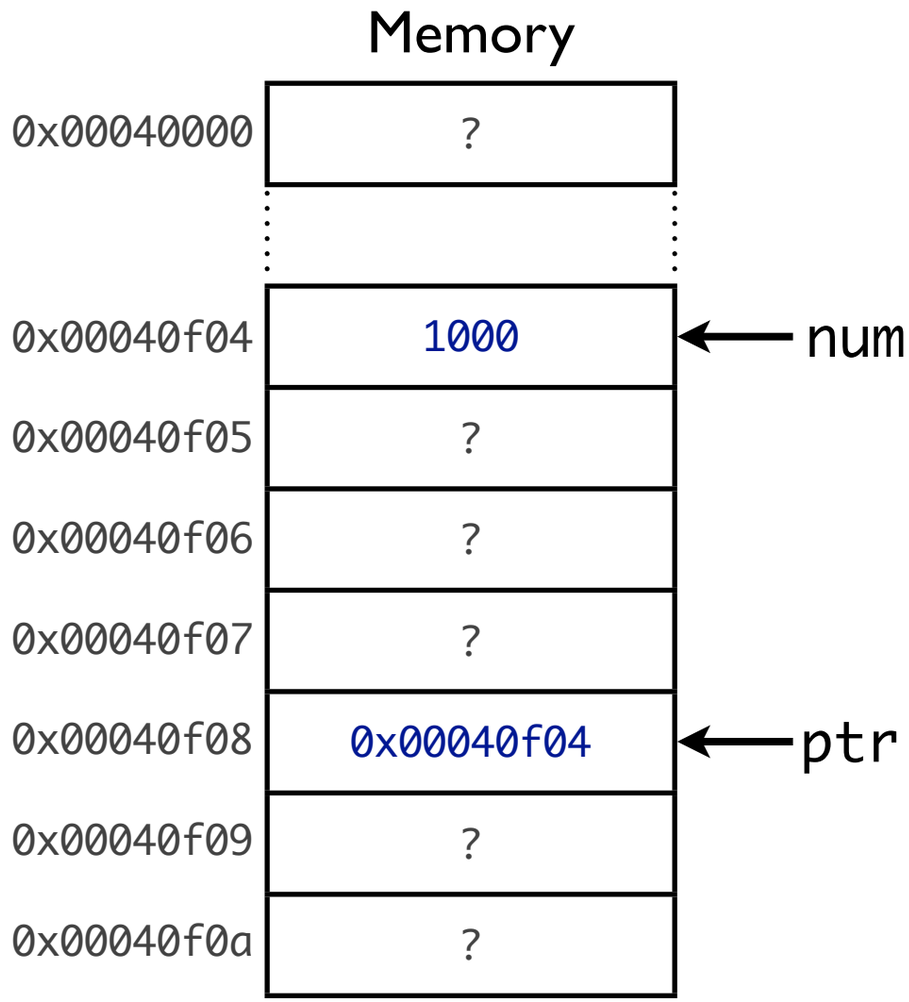
```
int num = 1000;
```



Pointers

Pointers are simply variables that store memory addresses

```
int num = 1000;

int* ptr = &num; // ptr = address of num
```



Pointers

Pointers are simply variables that store memory addresses

```
int num = 1000;

int* ptr = &num; // ptr = address of num
```

Printing the values stored in these variables:

```
cout << num << endl;    // displays: 1000

cout << ptr << endl;    // displays: 0x00040f04
```

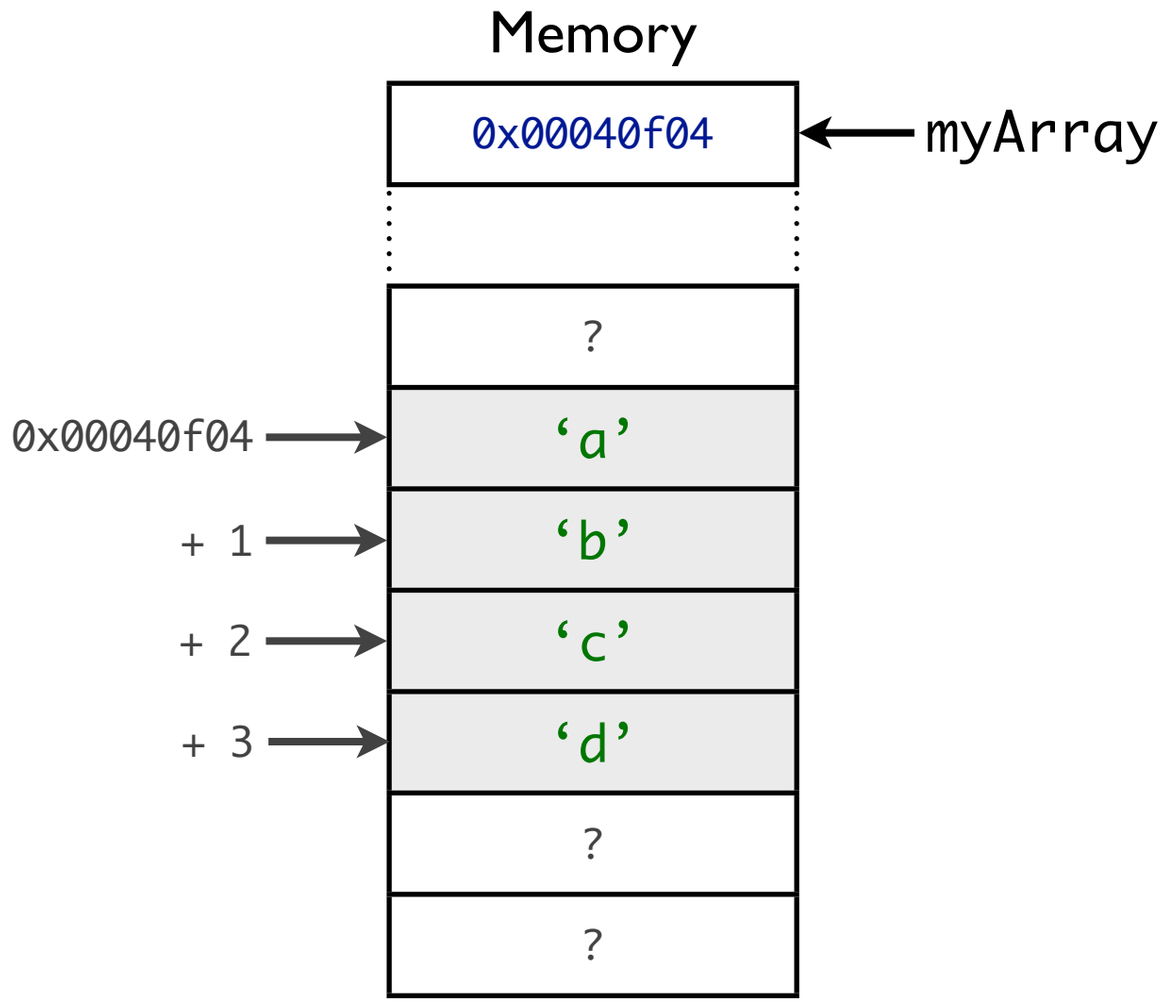
We can use `ptr` to get the value of `num` (this is called "dereferencing"):

```
cout << *ptr << endl;   // displays: 1000

cout << ptr[0] << endl; // displays: 1000

// second version is the same as saying *(ptr + 0)
```

Array variables are pointers!

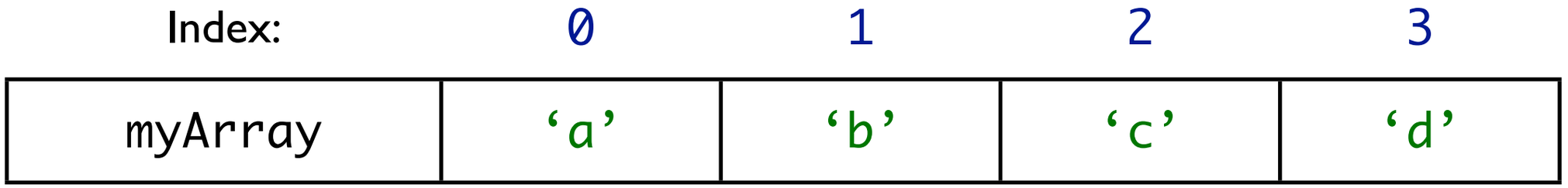
Pointers are simply variables that store memory addresses

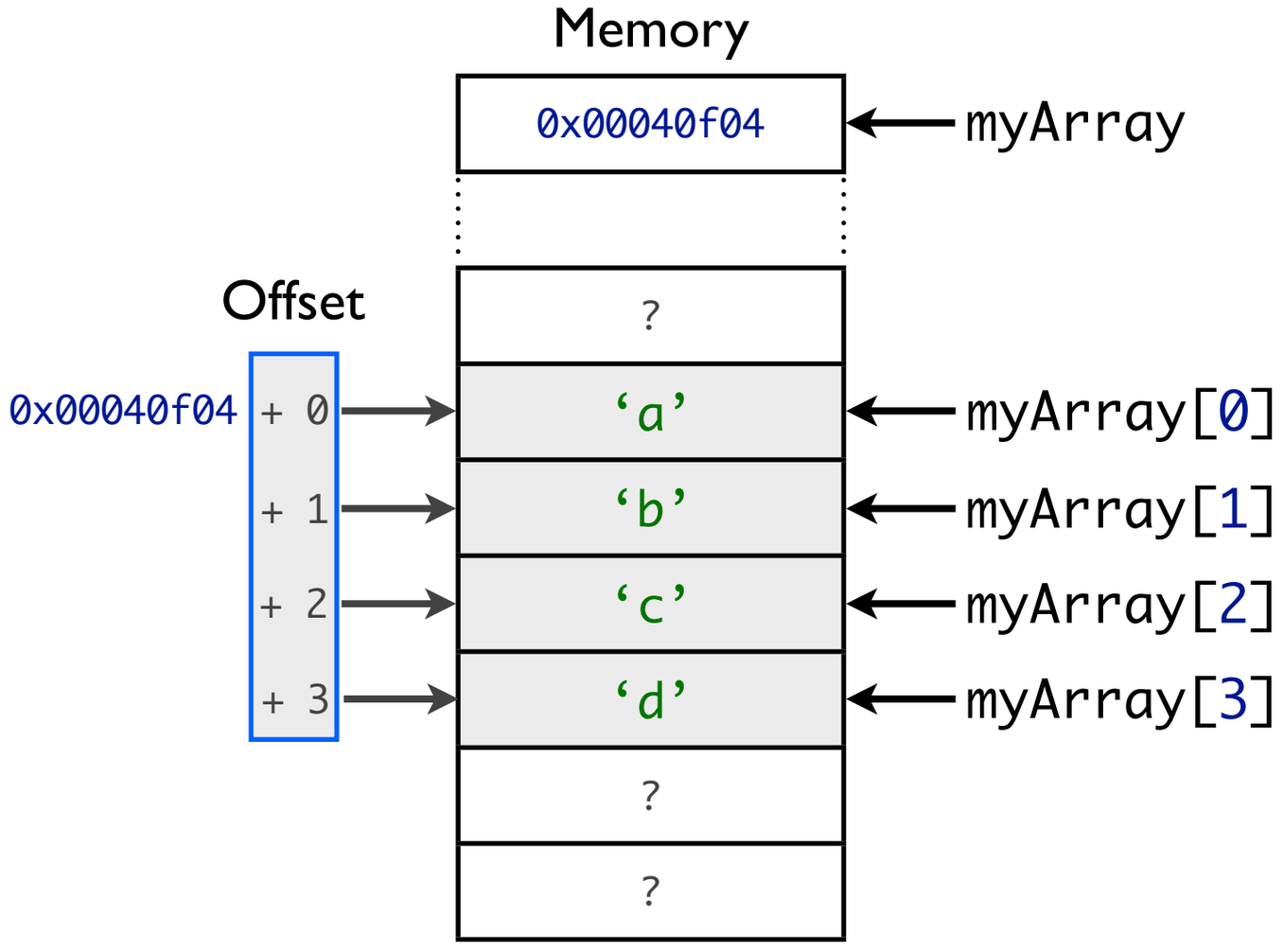
```
// myArray is just a pointer to the first element!
char myArray[] = {‘a’, ‘b’, ‘c’, ‘d’};
```



Array Indexes / Offsets

Each element in the array has a corresponding *index*:

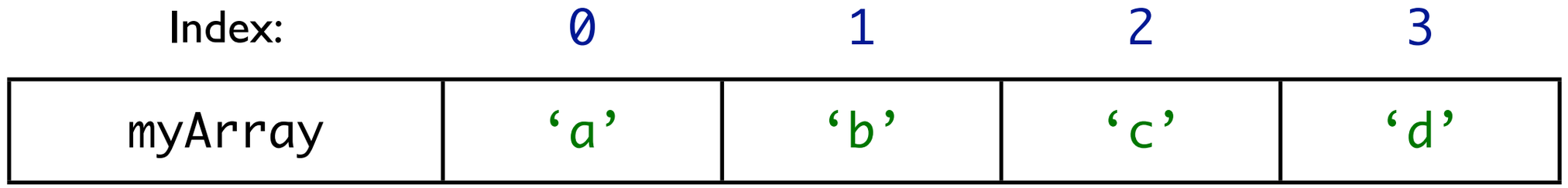


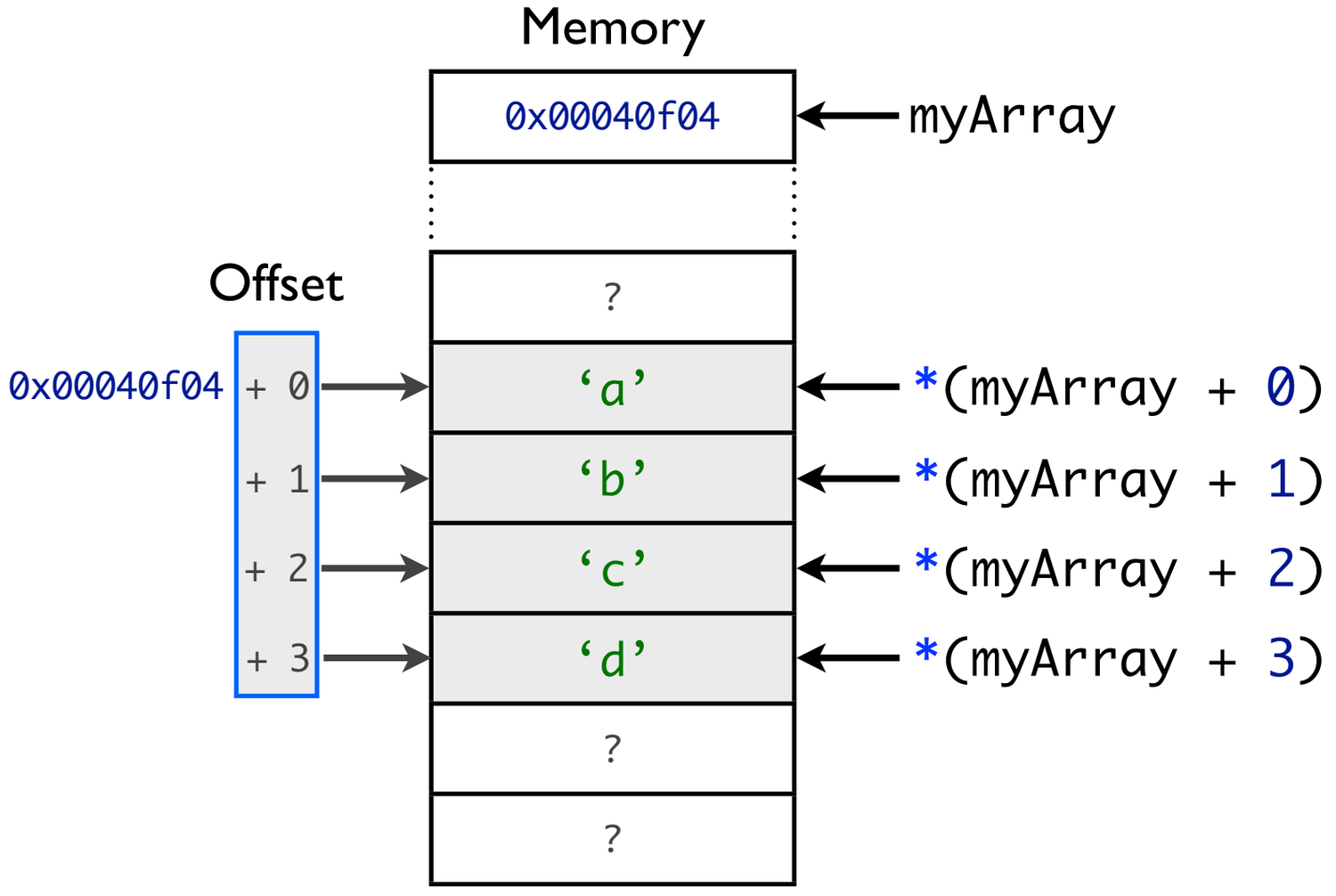


An element's index is the same as its offset!

Array Indexes / Offsets

Each element in the array has a corresponding *index*:



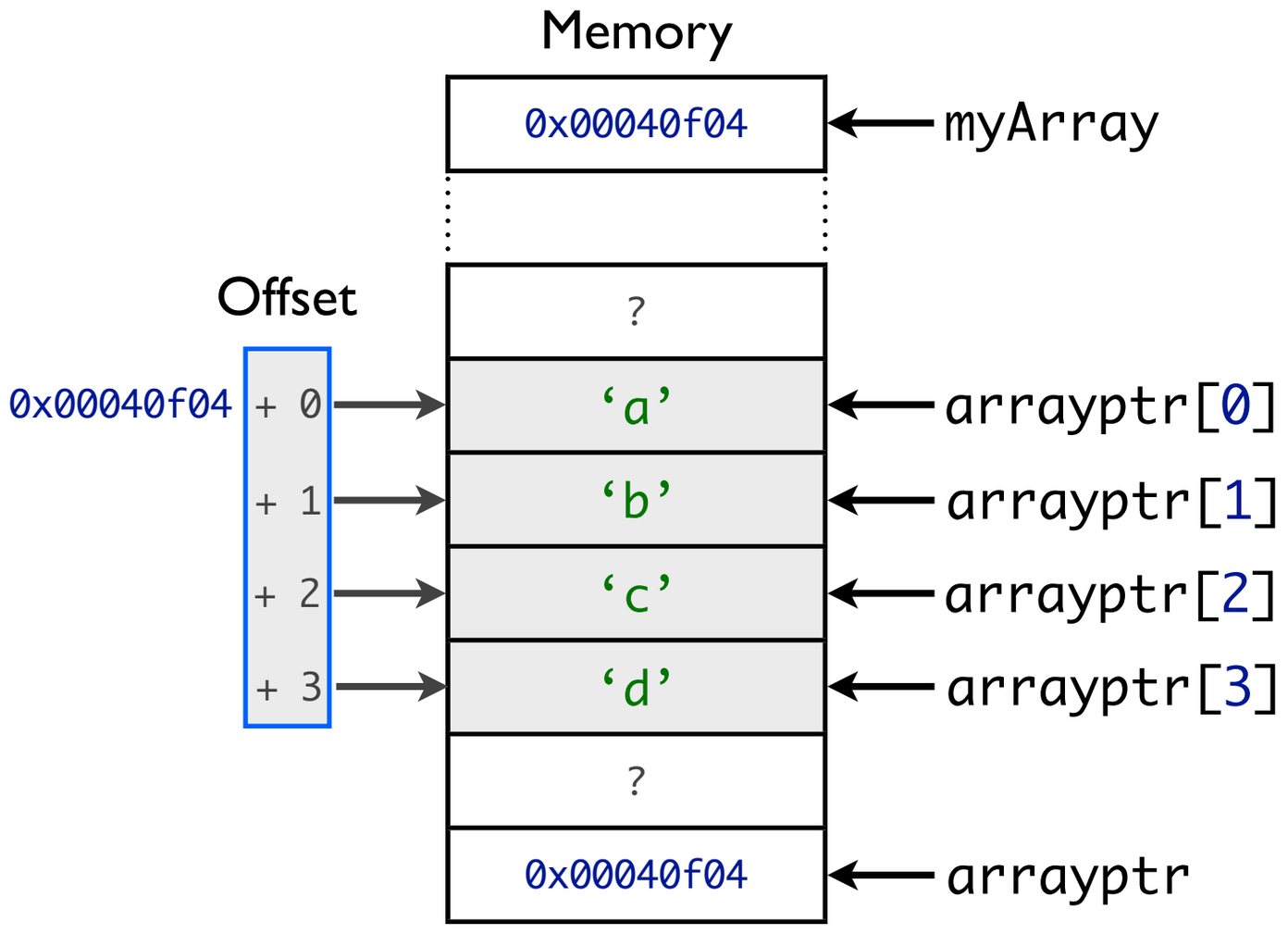


An element's index is the same as its offset!

Array variables are pointers!

Pointers are simply variables that store memory addresses

```
char myArray[] = {'a', 'b', 'c', 'd'};
char* arrayptr = myArray; // copy myArray value to arrayptr
```



Don’t be scared of pointers!

(they’re simply variables that store memory addresses)

Don’t be scared of pointers!

(we’ll only use them in the context of dynamic arrays)

Be scared of vampires.

Be scared of grizzly bears.

Be scared of missing puzzle pieces.

Be scared of free candy?

Be scared of 8-bit Dr. Horrible

HA HA HA HA HA HA HA HA HA HA...

(be sure to watch the real one, though)

HA HA HA HA HA HA HA HA HA HA...

Seriously! It's good stuff.

Don’t judge me!

For better or for worse, I am who I am…

Don’t be scared of pointers!

(they’re simply variables that store memory addresses)

Dynamic Memory Allocation

So far, we've been using statically-allocated variables

- the compiler allocates memory before the program is even run
- additionally, array sizes must be known beforehand (not always possible)

Dynamic memory allows us to allocate memory at run-time

- you can create arrays with arbitrary sizes (very useful)
- can potentially introduce *memory leaks* if not used properly

In this class, we're going to focus on dynamically allocated arrays

- after they're created, they work just like normal arrays
- they must be deallocated once they're no longer needed

The new (nothrow) operator

Dynamic memory allocation is done using the new operator

- new allocates memory for the specified variable
- it also calls any appropriate constructors
- if successful, a pointer to the newly created value is returned
- on failure, new (nothrow) returns `NULL` (a symbolic constant defined in `cstdlib`)

Examples:

```
// creates an array of 100 integers
int* int_ptr = new (nothrow) int[100];
// creates an array of a million strings
string* str_ptr = new (nothrow) string[1000000];
// creates an array of ‘count’ Triangle objects
Triangle* triangles = new (nothrow) Triangle[count];
```

The new (nothrow) operator

Computers have a finite amount of memory available

- if you try to allocate more space than is available, the operator will fail
- new (nothrow) returns NULL if the allocation was unsuccessful
- you should test newly allocated pointers to ensure that memory was properly allocated

Allocating memory and checking for errors:

```
// creates an array of strings
string* word_array = new (nothrow) string[50000];


// check if allocation was successful; print an error and exit, if not
if (word_array == NULL) {
    cout << “Not enough memory for 50,000 strings!” << endl;
    exit(1);
}
```

Dynamically Allocated Arrays

Example of allocating memory for an array of integers:

```
// array_size is an int variable whose value is not known at compile-time
int array_size;


// dynamically allocate space for an array of ‘array_size’ integers
int* array = new (nothrow) int[array_size];

// if the allocation failed, print an error message and exit
if (array == NULL) {
    cout << “Not enough memory for ” << array_size << “ integers!” << endl;
    exit(1);
}
```

Dynamically Allocated Arrays

After you allocate the memory for an array, use it like normal:

```
int* array = new (nothrow) int[num_values];


// assign each element a value
for (int i = 0; i < num_values; i++) {
    array[i] = i * 1000; // just like normal!
}

// prints each element in the array to the console
for (int i = 0; i < num_values; i++) {
    cout << array[i] << endl; // just like normal!
}
```

Deallocating Dynamic Memory

Statically allocated variables are automatically cleaned up by C++

- such variables are created when entering the scope in which they exist and are destroyed (their memory deallocated) when leaving

Dynamic memory is not auto-deallocated; we must do it ourselves

- use the delete operator to deallocate a single value (we won't use this!)
- use the delete[] operator to deallocate an entire array (we WILL use this)

Since we're only using dynamic allocation in the context of arrays,

- you will only ever need to use delete[] in this class

General syntax:

```
delete [] array; // frees all memory used by 'array'
```

Deallocating Dynamic Memory

Example of deallocating a dynamic-memory array using `delete[]`:

```
// dynamically allocate space for an array of 10 doubles
double* dm_array = new (nothrow) double[10];


// if the allocation failed, print an error message and exit
if (dm_array == NULL) {
    cout << “Not enough memory for 10 doubles!” << endl;
    exit(1);
}


// deallocate the space used by the array
delete [] dm_array;
```

const and pointers

Remember that const has a *trillion* different uses? Here are more!

```
// ‘p’ is a pointer to a constant int variable; the object
// cannot be modified via the pointer
int const* p;

// ‘p’ is a constant pointer to an int variable; you cannot
// change the pointer, but you can change the variable via p
int* const p;

// ‘p’ is a constant pointer to a constant int variable; you
// can’t change the pointer, nor can you change the value via p
int const* const p;
```

const and pointers

These are easy to remember if you read them *right-to-left*:

```
string const* s;
```

Like this:

```
        s          *          const        string
   "s is a"  "pointer to a"  "constant"  "string object"
```

Try interpreting these:

```
string const* const str;
```

```
string* const str;
```

Using const when appropriate is encouraged, but you won't be graded on this.

Dynamic Memory Recipe

When working with dynamic memory, you should always:

- declare a new pointer variable with the correct data type, initialized with the address returned by the new (nothrow) operator
- check if the pointer equals NULL. If it does, then memory allocation failed; display an error message and exit
- use the memory
- free the memory with the delete[] operator when it is no longer needed

Make sure that you:

- only use delete[] on memory addresses that were returned by the new operator (anything else represents an error)
- only delete[] allocated memory once; additional attempts will result in errors

Watch it if you get bored. =)

SENSE

This picture makes none