Arrays

Hip, hip, array!

One Reference to Rule Them All

In our programs so far:

- each variable referred to only one value
- we had to have a new variable for each value we were using

Enter the future:

- ARRAYS (huzzah)!
- arrays let us use a single identifier to reference *multiple values*

Your life is about to get better.

- I'll prove it to you with a demo. =)

What is an Array?

An array is:

- a collection of similar data
- a location in memory with space for multiple variables of the same type
- almost as good as ice cream. *Almost.*

Features of arrays:

- arrays allow us to use a single identifier to reference multiple data values
- we can easily iterate (loop) over arrays
- arrays allow both random and sequential access to the data they contain
- arrays and similar data structures are incredibly common in most programs

Declaring an Array

General syntax:

```
// declares an array of ‘type’ with ‘SIZE’ elements
type identifier[SIZE];
```

Notice the various components:

- type: specifies the data type of the elements in the array (e.g. int, double, bool, string, etc)
- identifier: the name of the variable (yes, an array is a variable, too!)
- `[SIZE]`: a pair of square brackets containing an *integer* value representing the number of elements in the array

Declaring an Array

General syntax:

```
// optionally, you can specify initial values
type identifier[SIZE] = {value1, value2, …, valueN};
```

Notice the various components:

- type: specifies the data type of the elements in the array (e.g. int, double, bool, string, etc)
- identifier: the name of the variable (yes, an array is a variable, too!)
- `[SIZE]`: a pair of square brackets containing an integer value representing the number of elements in the array
- `{…}` an optional initialization list specifying values for some or all of the array elements
  - if this is provided, then `SIZE` may be omitted

Declaring an Array

The size of an array can be specified:

- explicitly, using an integer literal:

```
double num_array[10];
```

- explicitly, using an integer constant (preferred):

```
const int SIZE = 10;
double num_array[SIZE];
```

- implicitly, using only an initialization list:

```
// creates an array of 5 ints
int num_array[] = {1, 2, 3, 4, 5};
```

no explicit size!

size is *implied* by # of elements

Declaring an Array

The size of an array can be specified:

- explicitly, while also providing initial values:

```
// creates an array of 10 ints
// the first 5 are specified, the rest are 0's
int num_array[10] = {1, 2, 3, 4, 5};
```

Declaring an Array

Using initializer syntax (the curly brackets):

- `{...}` an optional initialization list specifying values for some or all of the array elements
- if the size of the initializations list is *less than the explicitly declared size*, then the remaining values will be whatever the default value is for the array's type
  - the default value for numeric types is zero
  - the default for strings is the empty string ("")

Examples:

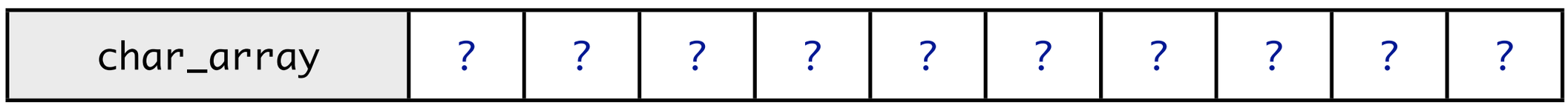
```
// an array containing ten 0's
int numbers[10] = { };


// an array containing five 2's and five 0's
int numbers[10] = {2, 2, 2, 2, 2};
```

Declaring an Array

Example:

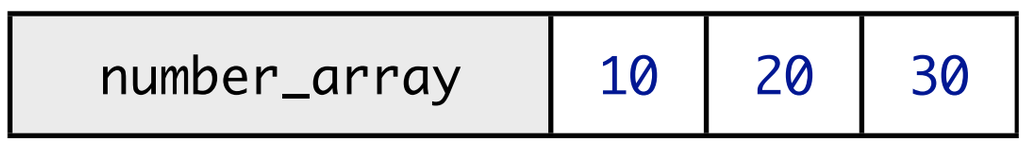
```
// declares an array of chars with 10 elements
// all 10 elements have garbage values
char char_array[10];
```



Declaring an Array

Example:

```
// declares an array of 3 ints, each w/ specific value
// size determined by number of elements provided
int number_array[] = {10, 20, 30};
```



Declaring an Array

Example:

```
// a constant specifying the size of the array
const int SIZE = 10; // or any positive integer

// declares an array of ints with ‘SIZE’ # of elements
// all elements initialized to 0
int number_array[SIZE] = {};
```

number_array	0	0	0	0	0	0	0	0	0	0

Declaring an Array

Example:

```
// a constant specifying the size of the array
const int SIZE = 10; // or any positive integer

// declares an array of ints with ‘SIZE’ # of elements
// provides 1st 3 elements with values, 0’s for rest
int number_array[SIZE] = {10, 20, 30};
```

number_array	10	20	30	0	0	0	0	0	0	0

Array Declaration Errors

Examples of INVALID declarations:

```
// the initialization list CANNOT be longer than the
// explicitly declared size
int sad_panda[3] = {10, 20, 30, 40};
```

```
// you MUST provide a size OR some initial values
int epic_fail[] = {};
```

Some Practice

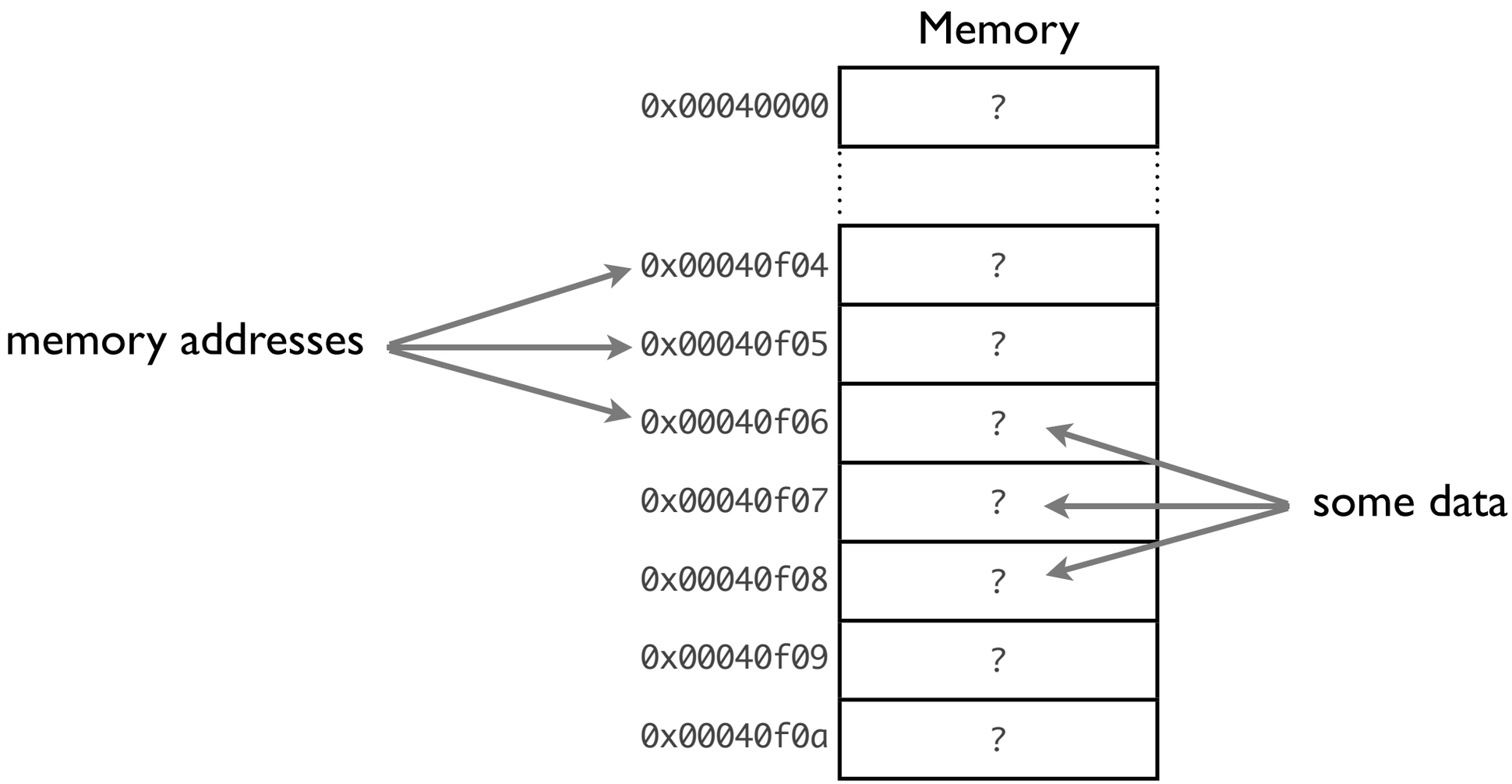
Write a single C++ statement that declares a(n):

- double array with space for 10 elements
- int array with element values -3, -4, and 100
- double array of 10,000 elements initialized to zero.
- char array with 32 implicitly declared values
- int array of SIZE elements (assume SIZE is a previously declared constant integral variable
- int array with SIZE elements and the first three elements equal to 1, 2, and 3.
- bool array with implicitly declared size and space for 7 elements.

A Walk Down Memory Lane

Computers store data 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

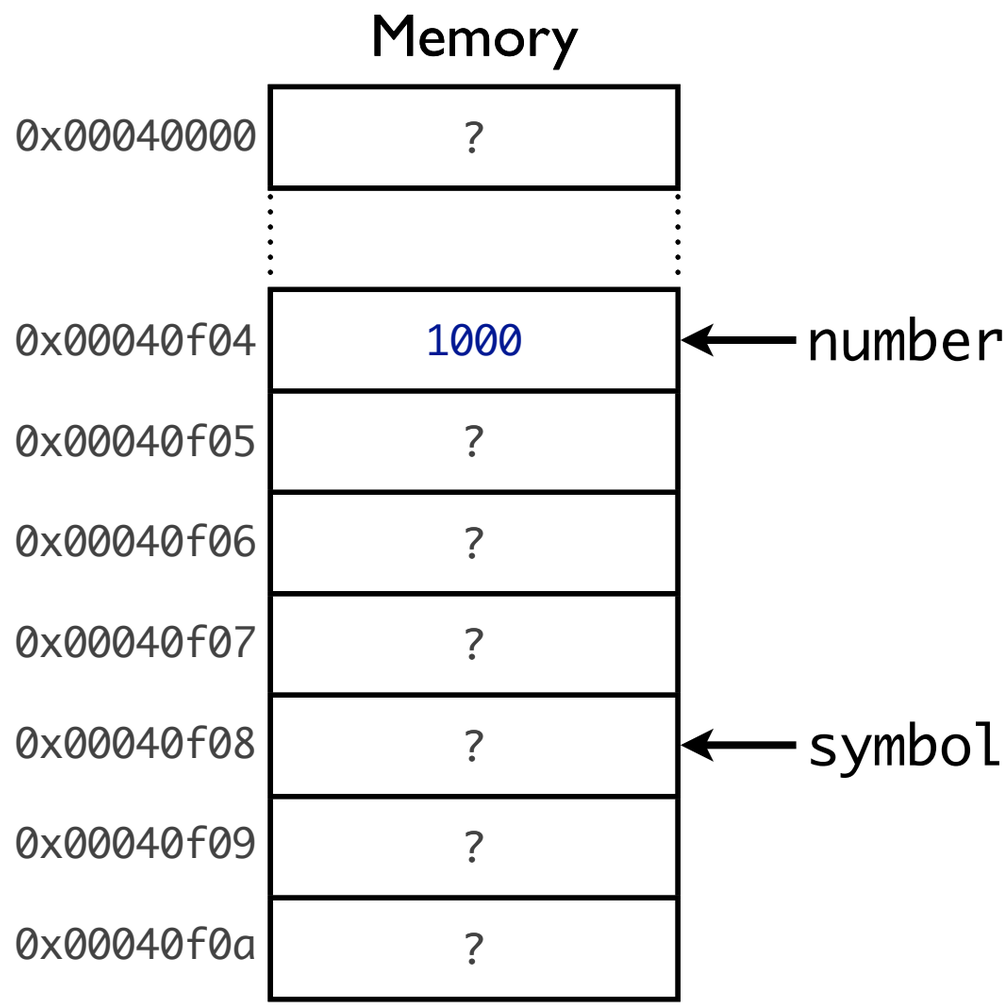


A Walk Down Memory Lane

Variables provide an easy way to reference memory locations

```
int number = 1000;

char symbol;
```



Arrays in Memory

Let's say you have this array declaration:

```
// creates a 4-element array of chars
char myArray[] = {'a', 'b', 'c', 'd'};
```

An array is stored as a *contiguous* block of memory

- the memory allocated to the array will be large enough to hold the specified number of elements (4 in the case of `myArray`)
- each element in the array will have the same data type

So, `myArray` is a block of 4 consecutive `char` values

- we can use it to reference multiple data values!
- this is really cool, because normal variables reference only a single value =)

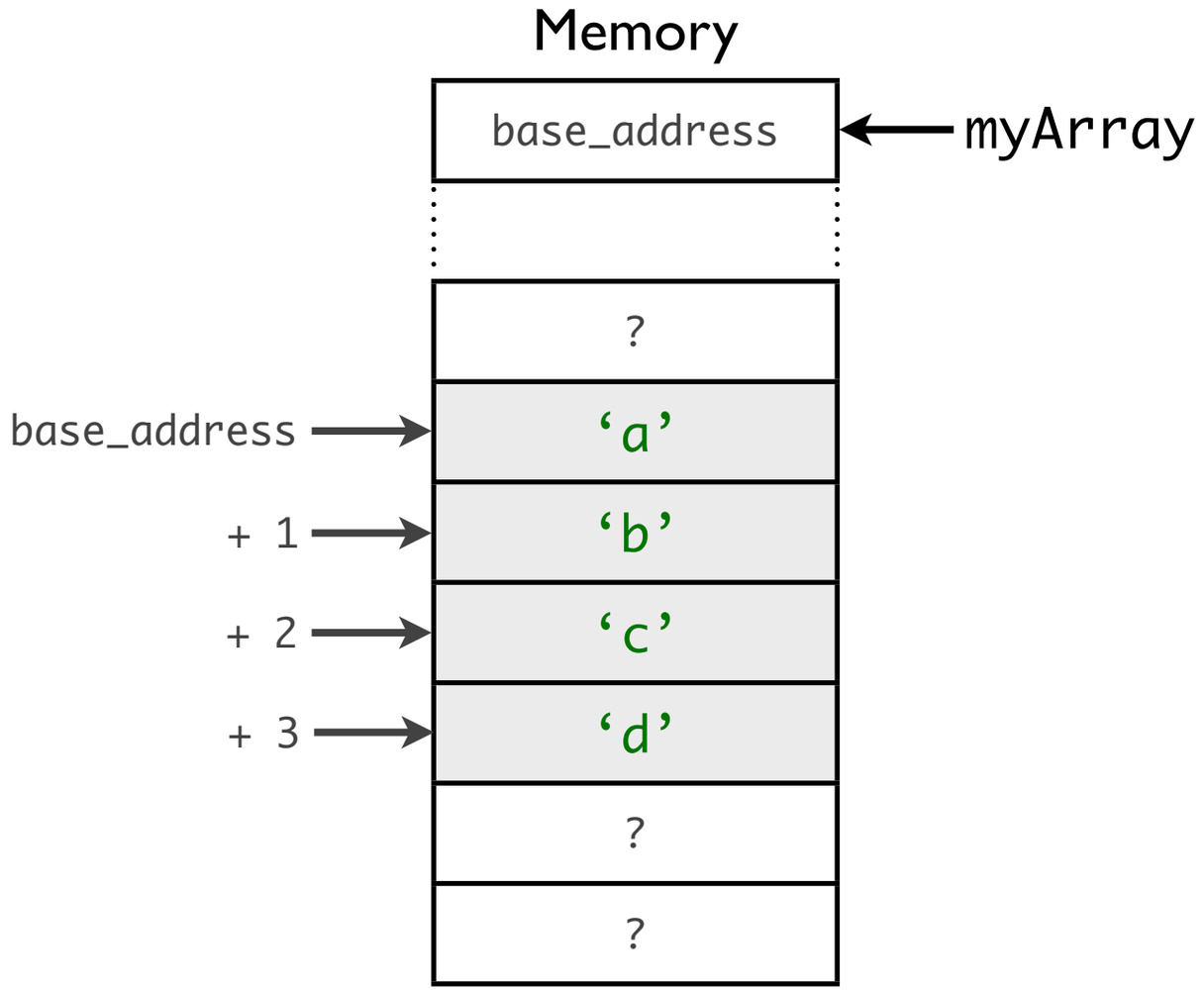
Arrays in Memory

Let's say you have this array declaration:

```
// creates a 4-element array of chars
char myArray[] = {'a', 'b', 'c', 'd'};
```

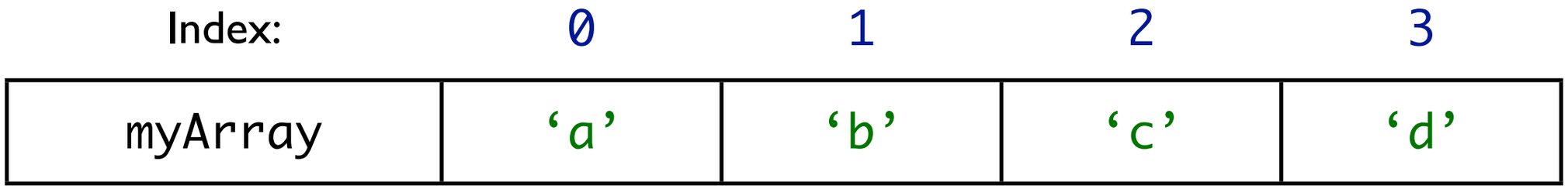
Behind the scenes:

- C++ allocates contiguous memory for 4 char values
- `myArray` gets the address of the first element (`base_address`)
- the first array element is stored at `base_address + 0`
- the second element is stored at `base_address + 1`
- and so forth...



Array Indexes

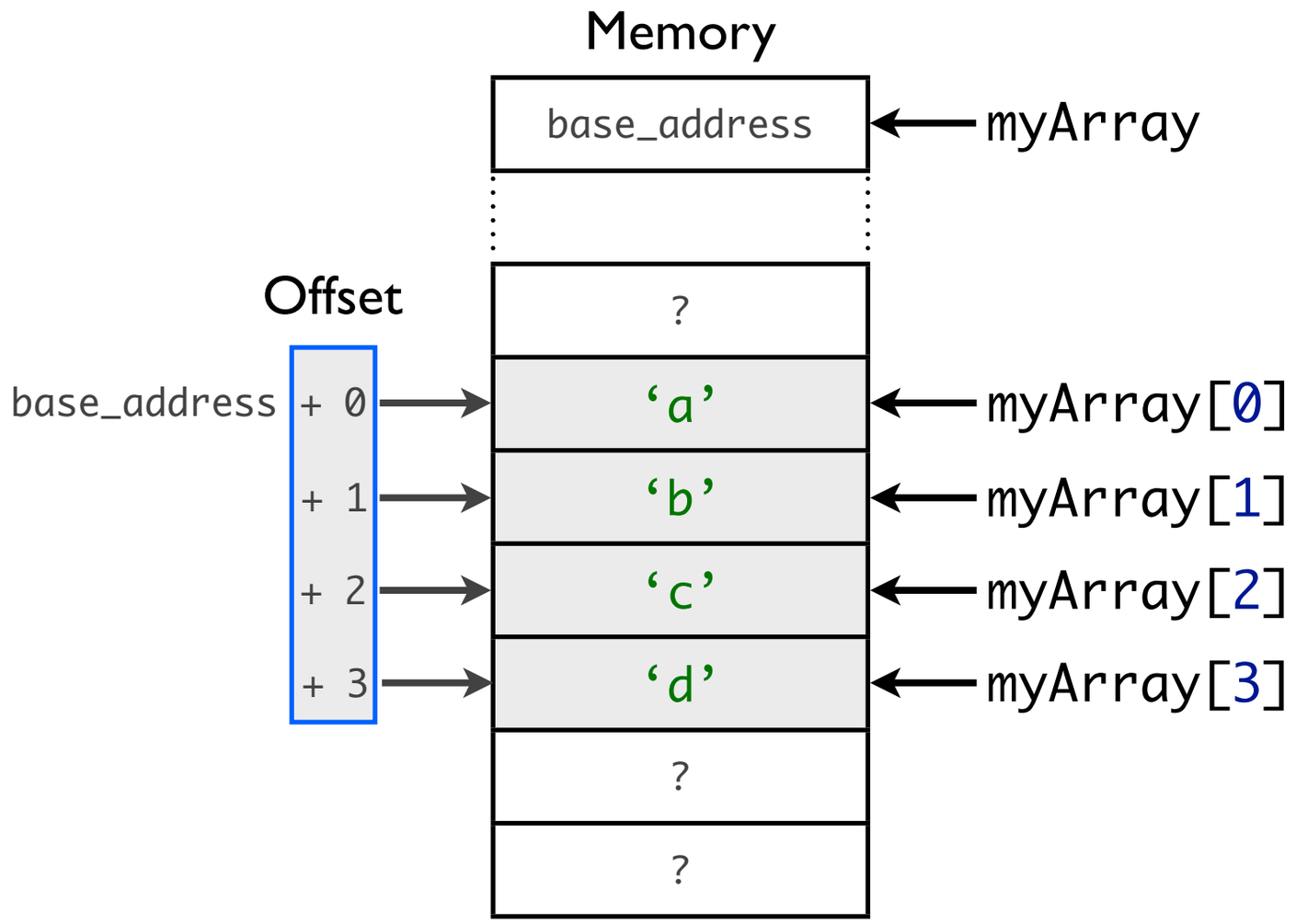
Each element in the array has a corresponding *index*:



Notice:

- array indexes start at ZERO!
- indexes are integer values
- element *index* and element *offset* are the same!

An element's index is used to access or modify that element's value!



Array Operator [ ]

An array is used to reference multiple data values

- we have to specify which element we want in the array
- this is done by using the element's index (or *offset*)

General Syntax:

```
// accesses the i'th element in 'myArray'

myArray[i];
```

Notes:

- `myArray` is the name of your array (obviously)
- `i` is any integer expression (e.g. an integer literal, integer variable, or a complex expression that evaluates to an integer)

Array Operator [ ]

An array is used to reference multiple data values

- we have to specify which element we want in the array
- this is done by using the element's index (or *offset*)

Examples using the *array operator*:

```
int x = 1, myArray[] = {1, 2, 3, 4, 5};

myArray[0]; // access the first element in myArray
myArray[x]; // access the second element
myArray[3*x + 1]; // access the last element
```

Accessing an Array Element

Let's say you have this array declaration:

```
// creates a 3-element array of doubles
double myArray[] = {10, 20, 30};
```

Use the *array operator* `[ ]` to access or modify an array element:

```
cout << myArray[0]; // outputs 10 (the first element)
cout << myArray[1]; // outputs 20 (the second element)
cout << myArray[2]; // outputs 30 (the third element)
```

Modifying an Array Element

You also use the array access operator to modify an element:

```
// change the first element (index 0) to 1000
myArray[0] = 1000;
```

You can read into an array element just like any other variable:

```
// stores user input into the second element (index 1)
cin >> myArray[1];
```

Modifying an Array Element

Let's say you have this array declaration:

```
// initialization list can be used when declaring the array
double myArray[] = {10, 20, 30};
```

You can only change one value at a time after the array is declared:

```
// initialization list CANNOT be used like this
myArray = {50, 100, 400}; // does not compile
```

Each array element must be changed individually:

```
myArray[0] = 50;
myArray[1] = 100;
myArray[2] = 400;
```

Iterating through an Array

Array indexes are consecutive integers, so for loops are ideal

```
// remember that array indexes go from 0 to SIZE-1
for (int i = 0; i < SIZE; i++) {
    myArray[i]; // the i'th element
}
```

Some notes:

- start the loop variable at 0
- use the size (or number of elements) in the array as the upper limit
- be sure to use the less-than operator (<) in the condition!

Iterating through an Array

Array indexes are consecutive integers, so for loops are ideal

```
// iterate through the array in reverse order
for (int i = SIZE-1; i >= 0; i--) {
    myArray[i]; // the i'th element
}
```

To go through the array in reverse:

- start the loop variable at `SIZE-1`
- use `(i >= 0)` as the condition
- and decrement the loop variable instead of incrementing it

Iterating through an Array

Array indexes are consecutive integers, so for loops are ideal

```
// number of elements in the array
const int SIZE = 10;

 // declares array of 'SIZE' doubles
double numbers[SIZE];

// assign values to each array element
for (int i = 0; i < SIZE; i++) {
    numbers[i] = i * 100; // use loop var for index!
}
```

Element	Value
numbers[0]	0.0
numbers[1]	100.0
numbers[2]	200.0
numbers[3]	300.0
numbers[4]	400.0
numbers[5]	500.0
numbers[6]	600.0
numbers[7]	700.0
numbers[8]	800.0
numbers[9]	900.0

Printing an Array

Let's say you have this array declaration:

```
// creates a 3-element array of doubles
double myArray[SIZE] = {10, 20, 30};
```

`myArray` actually stores the *memory address* of its first element!

```
// don't do this!
cout << myArray; // outputs 0xbfffe94c (a memory address!)
```

The right way to print an array:

```
for (int i = 0; i < SIZE; i++) {
    cout << myArray[i] << endl; // the i'th element
}
```

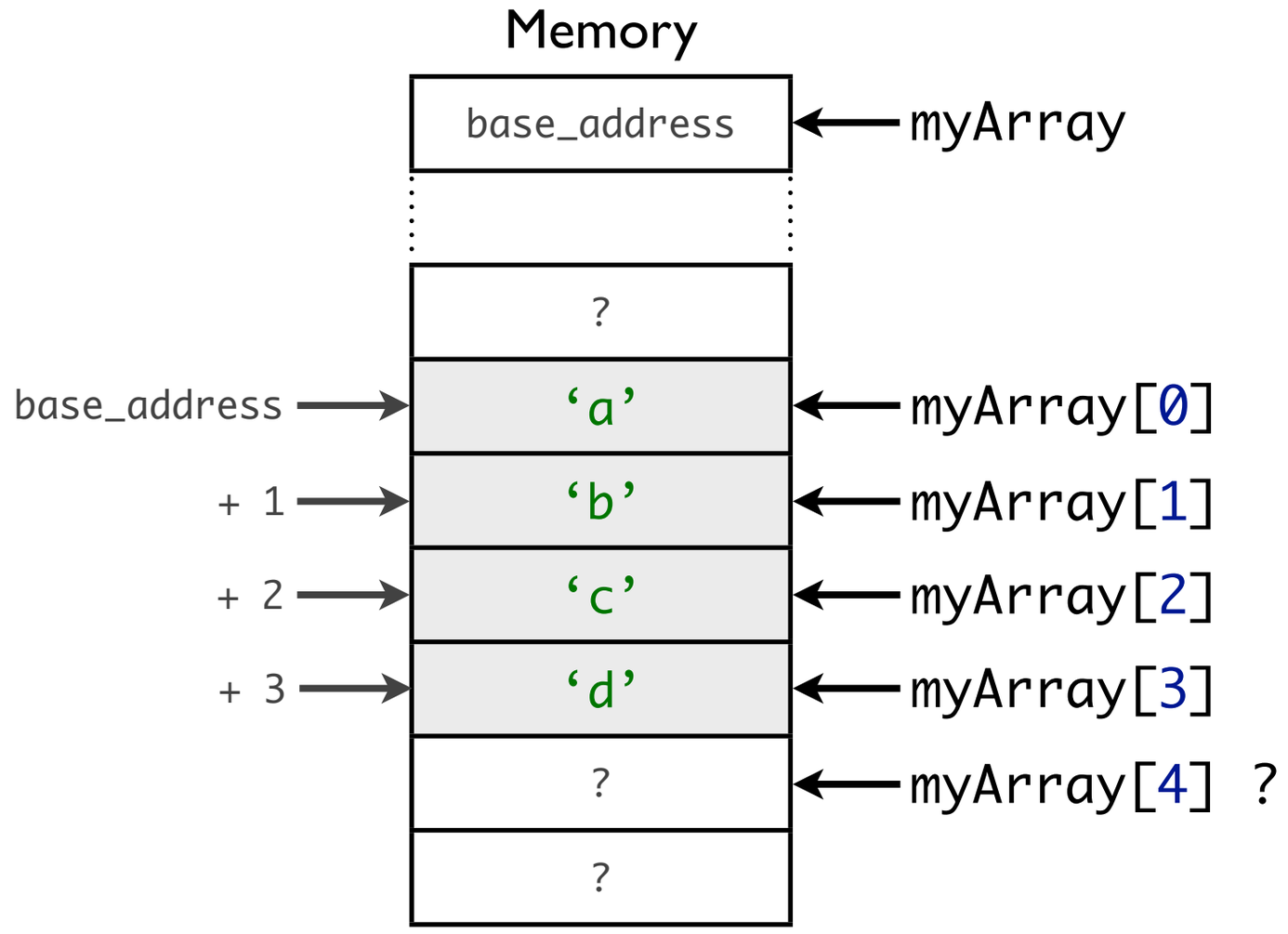
Out-of-Bounds Indexes

Let's say you have this array declaration (again):

```
// creates a 4-element array of chars
char myArray[] = {'a', 'b', 'c', 'd'};
```

What happens if you try to access `myArray[4]`?

- C++ doesn't check if the index is out of bounds!
- the memory before and after an array can be inadvertently accessed by using an index that is out-of-bounds…
- this can be other variables in your program or memory addresses that are not part of your program (*segfault*)



If your program is crashing,

I'll bet you a burrito it's because
of out-of-bounds array access.