Slides by Andrew DeOrio
and James Juett

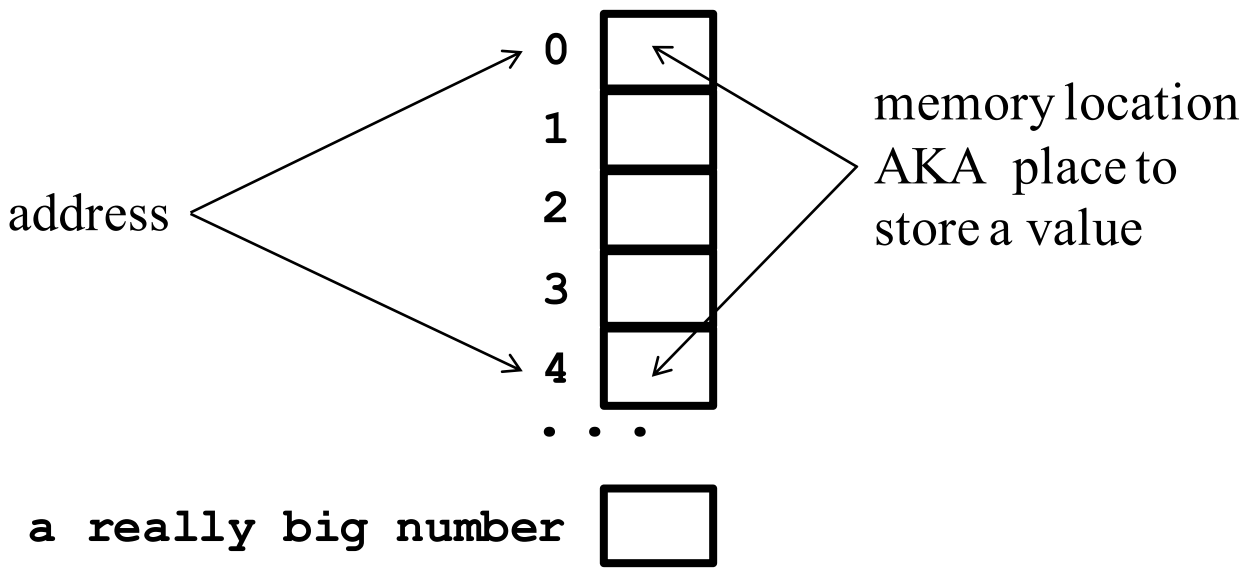
EECS 280

Programming and Introductory Data Structures

Pointers and Arrays

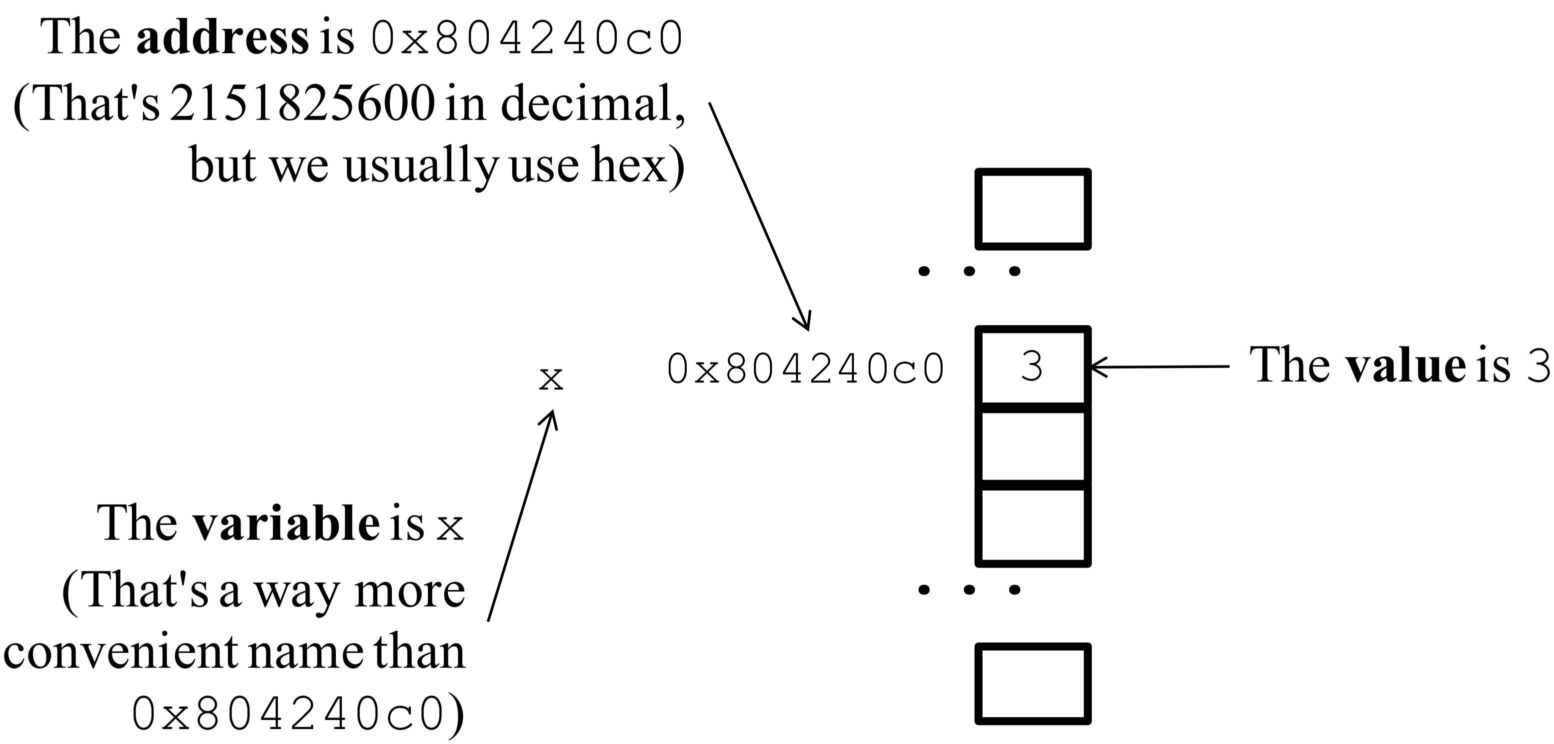
Review: memory

- Objects are stored in **memory**
- Memory is a bunch of storage locations numbered with **addresses** from 0 to a very large number
- The computer needs a way to find each **object**
- Each object lives in memory at an **address**



Review: memory

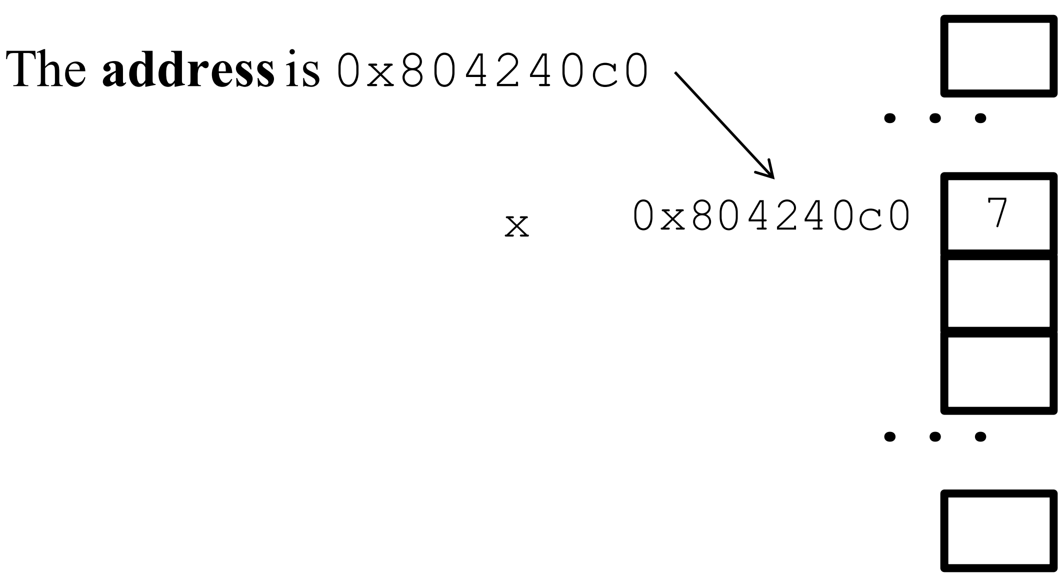
```
int x = 3;
```



Review: memory

- To get the address of a variable, use the *address of* operator

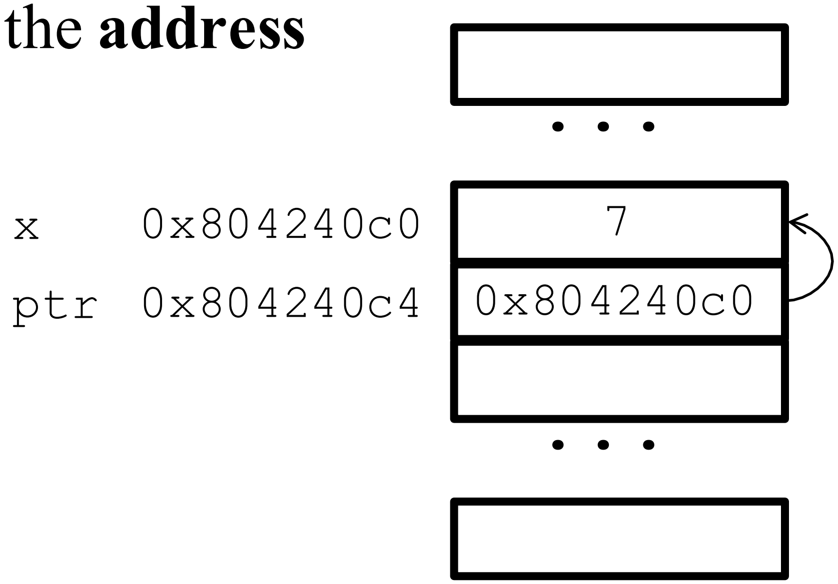
```
int x = 3;
x = 7;
cout << &x; //0x804240c0
```



Review: pointers

```
int x = 3;
cout << &x; //0x804240c0
int *ptr = &x;
```

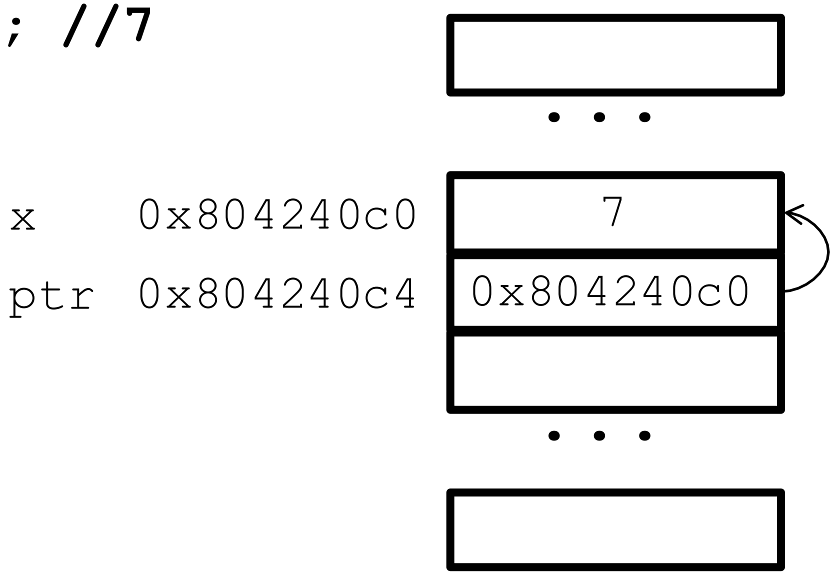
- A pointer is a type of object whose **value** is the **address** of an object



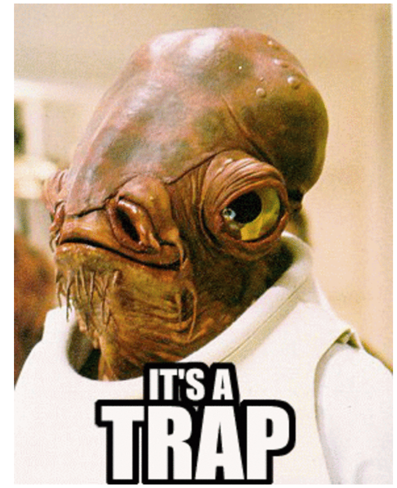
Review: dereference operator

- To get the object a pointer points to, use the `*` operator
  - Pronounced as "dereference" or "indirection"
  - "Follows" the pointer to its object

```
int x = 7;
int *ptr = &x;
cout << *ptr; //7
```



Declarations vs. expressions



Declaration

- `int *ptr;`
  - * means a pointer type
  - "ptr is a pointer to int"

- `int &ref;`
  - & means a reference type
  - "ref is a reference to int"

Expression

- `cout << *ptr;`
  - * means dereference operator
  - Follow pointer

- `cout << &ptr;`
  - & means address-of operator

Two different meanings for * and & depending on the context

Pointer practice

```
void add_one(int *x){
 *x += 1;
}
```

```
int main(){
  int a = 42;
  int *p = &a;
  cout << a;
  cout << *p;
  add_one(p);
  cout << a;
  cout << *p;
```

```
  //main continued ...
  add_one(&a);
  cout << a;
  cout << *p;
  int *p2 = p;
  add_one(p2);
  cout << a;
  cout << *p;
  return 0;
}
```

Review: arrays

- An **array** is a **contiguous** chunk of memory holding a sequence of objects of the **same type**

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

array	array[0]	0x804240c0	1
	array[1]	0x804240c4	2
	array[2]	0x804240c8	3
	array[3]	0x804240cc	4
	array[4]	0x804240d0	5

When arrays turn into pointers

- An array has no value
- If you try to look up the value of an array, you get a **pointer to the first element** instead

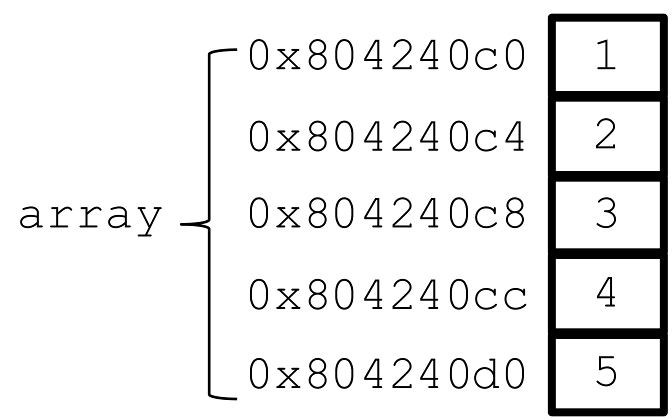
```
int array[5] = {1,2,3,4,5};
cout << array; //0x804240c0
```

array	array[0]	0x804240c0	1
	array[1]	0x804240c4	2
	array[2]	0x804240c8	3
	array[3]	0x804240cc	4
	array[4]	0x804240d0	5

When arrays DON'T turn into pointers

- As long as you don’t use an array where a value would be expected, it doesn’t turn into a pointer
- Example: the address-of "&" operator
  - `&arr` gives you a pointer to the whole array
  - Type of `&arr` is `int (*)[5]` – a pointer to an array of 5 ints
  - Not the same as turning into a pointer to first element

```
int array[5] = {1,2,3,4,5};
cout << array; //0x804240c0
cout << &array; //0x804240c0
```

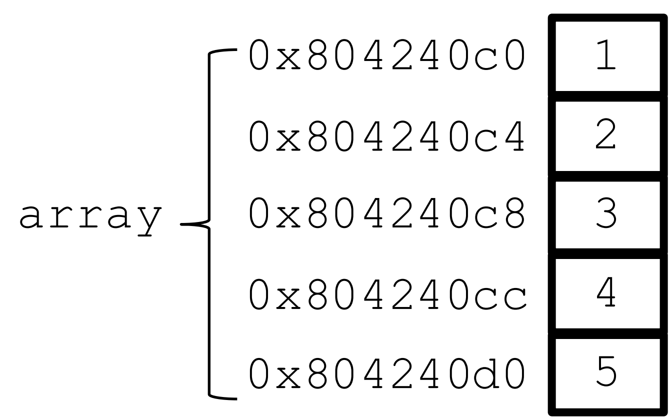


When arrays DON'T turn into pointers

- As long as you don’t use an array where a value would be expected, it doesn’t turn into a pointer
- Example: the `sizeof()` operator
  - Returns the size in bytes of an object or type

```
int array[5] = {1,2,3,4,5};
cout << sizeof(array); //20
```

- Why 20?



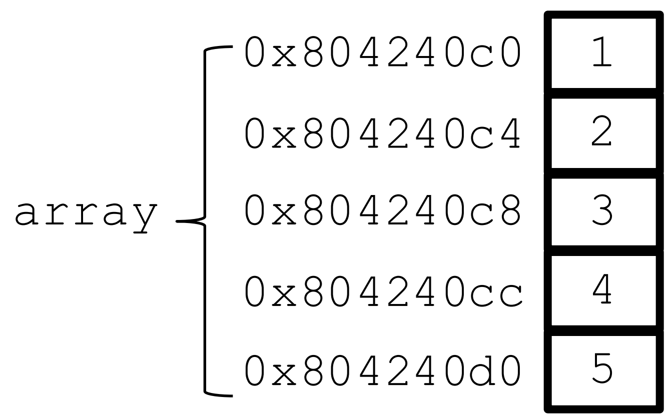
The sizeof() operator

- `sizeof()` returns the size in bytes of an object or type

```
int array[5] = {1,2,3,4,5};
cout << sizeof(array); //20
```

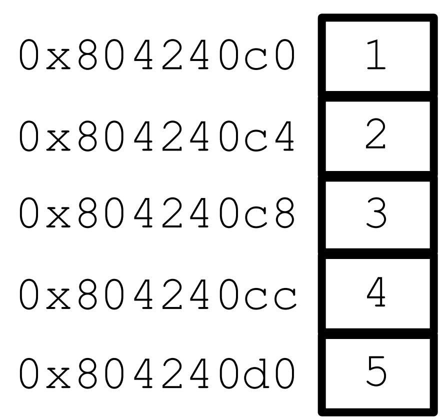
- Why 20? Because:

```
cout << sizeof(int); //4
```



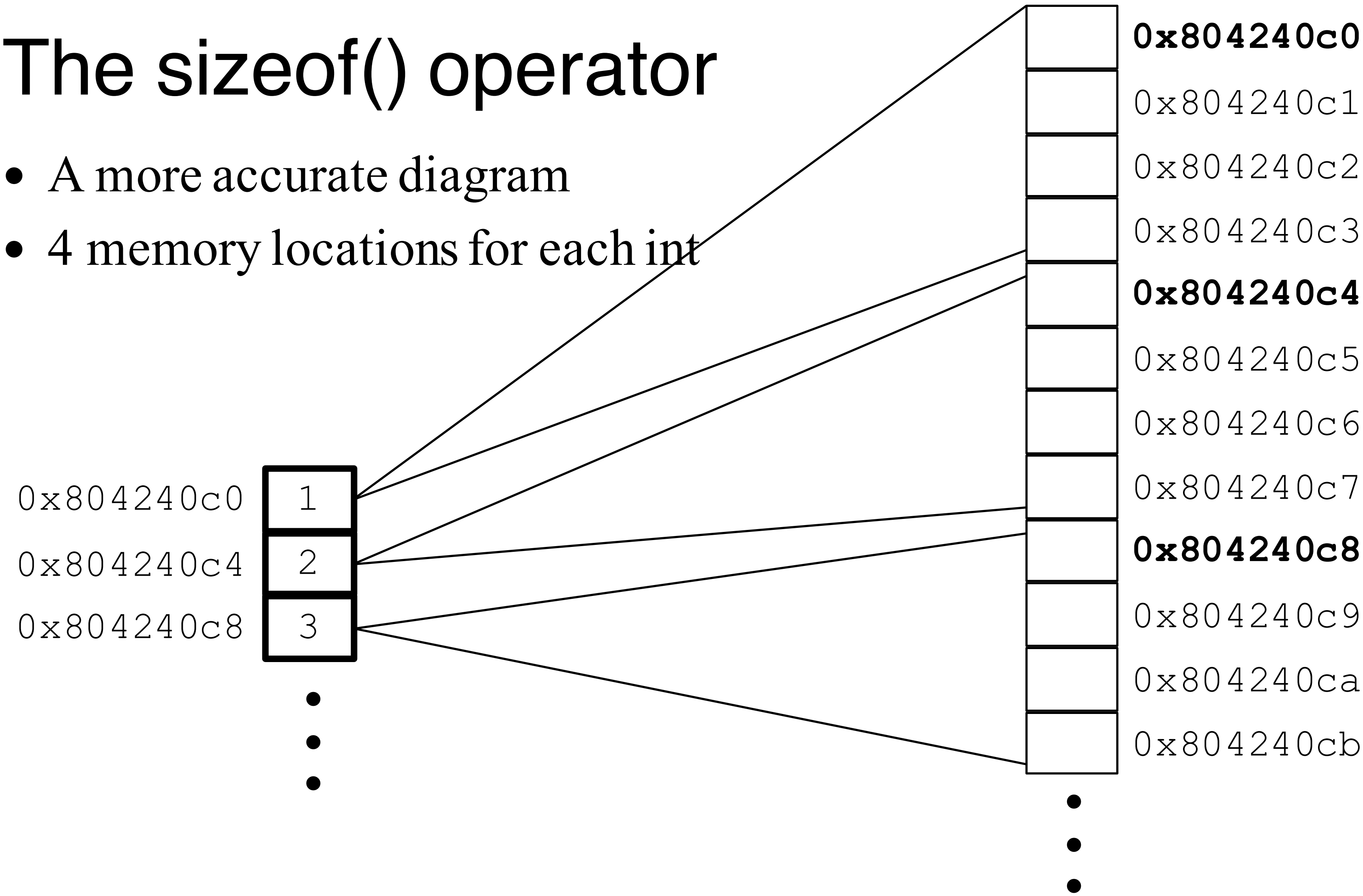
The sizeof() operator

- Each memory location holds one byte
  - Notation: 1 B
- An `int` occupies 4 bytes (on this machine)
  - 1 B = 8 bits, so 4 B = 32 bits
- That's why these addresses are spaced "by fours"



The sizeof() operator

- A more accurate diagram
- 4 memory locations for each int



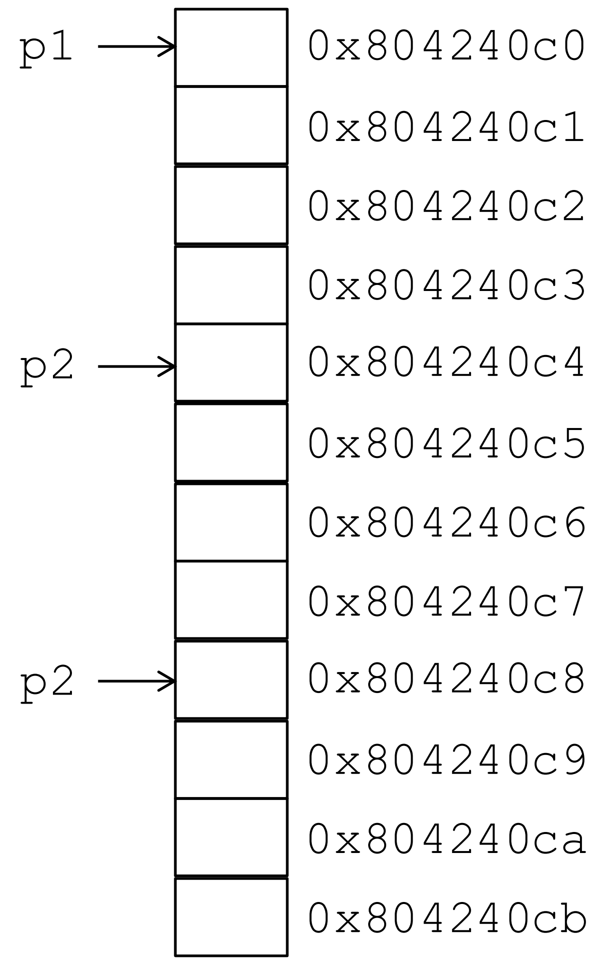
Type sizes

- The amount of memory assigned to a data type is a source of innumerable “portability bugs” in programs
- For example, suppose someone writes a program that assumes that all \ints are 8 bytes long. If that program is compiled on a 4-byte-int compiler, it is likely to break, if it compiles at all.
- There are **some** guarantees, however:
  - A “char” is always one byte
  - A “short” is always at least as big as a char
  - An int is always at least as big as a short
  - A long is always at least as big as an int
- However, while a “char” is always one byte, the restrictions on byte are strange: it must have **at least** eight bits, but could have more!

Pointer arithmetic

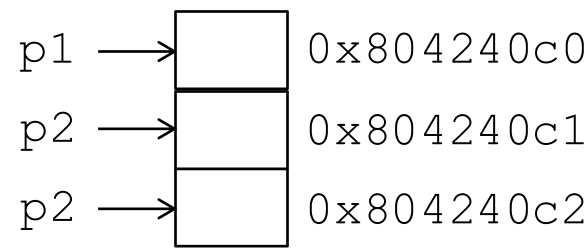
- `int array[3] = {1,2,3};`
- `int *p1 = array;`
- `int *p2 = array + 1;`
- `int *p3 = array + 2;`

- Why?
- Because `sizeof(int) == 4`
- The compiler knows that `p1` is a pointer-to-int
- "`p1 + 1`" means "the next int"



Pointer arithmetic

- `char array[3]={'a','b','c'};`
- `char *p1 = array;`
- `char *p2 = array + 1;`
- `char *p3 = array + 2;`



- Why?
- Because `sizeof(char) == 1`
- The compiler knows that `p1` is a pointer-to-char
- "`p1 + 1`" means "the next char"

Pointer arithmetic

- Pointer arithmetic
  - `int *ptr;` The compiler knows how big an `int` is
  - `ptr + x` computes the address `x` `int`s forward in memory
  - Operators: `+, -, +=, -=, ++, --`

- We can also use comparison operators with pointers
  - `<, <=, >, >=, ==, !=`
  - These just compare the address values numerically

- Warning! Pointer arithmetic only makes sense in arrays!
  - Arrays are guaranteed to be **contiguous** memory

Back to array size

- An "array pointer" is just like any other pointer when the program is running
- It doesn't know anything about the array it came from
- We have to be careful…

Where does an array end?

- What happens if a pointer wanders outside of its array?

```
int array[5] = {1,2,3,4,5};
cout << array[42];
```

  - Undefined behavior!
  - You end up reading/writing random memory
  - Program might crash, or maybe not
    Or maybe only sometimes ← Horrifying!
  - Anything can happen, *including getting the right answer*

- How do we keep pointers inside their arrays?
  - **Keep track of the length separately**
  - Put a sentinel value at the end of the array

Traversal by index

- Traversal by Index
  - Keep track of an integer **index** variable
  - To get an element, use the index as an **offset** from the beginning of the array

```
int const SIZE = 5;
int array[SIZE] = {1, 2, 3, 4, 5};

for(int i=0; i < SIZE; ++i){
  cout << array[i] << endl;
  cout << *(array + i) << endl; //same thing
}
```

Traversal by pointer

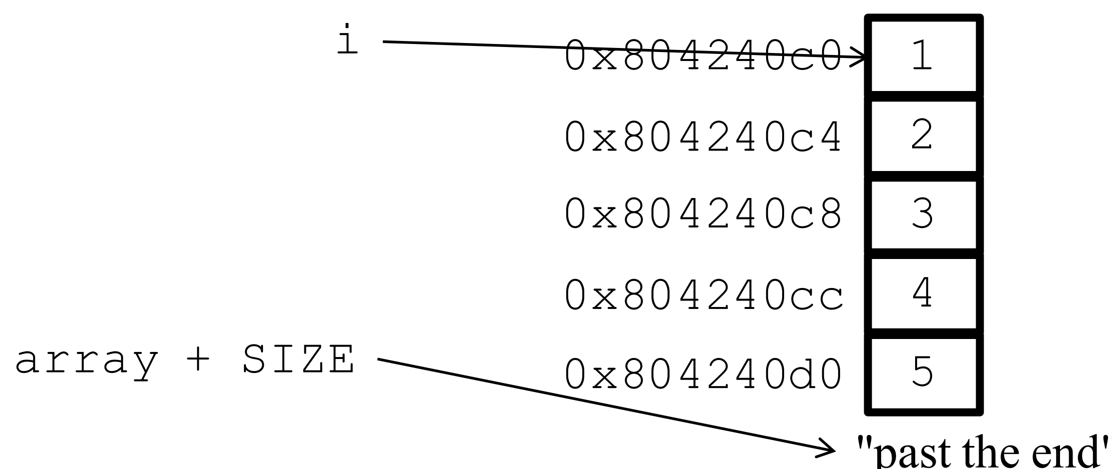
- Traversal by Pointer
  - Walk a **pointer** across the array elements
  - To get an element, just dereference the pointer

```
int const SIZE = 5;
int array[SIZE] = {1, 2, 3, 4, 5};

for(int *i=array; i < array + SIZE; ++i){
  cout << *i << endl;
}
```

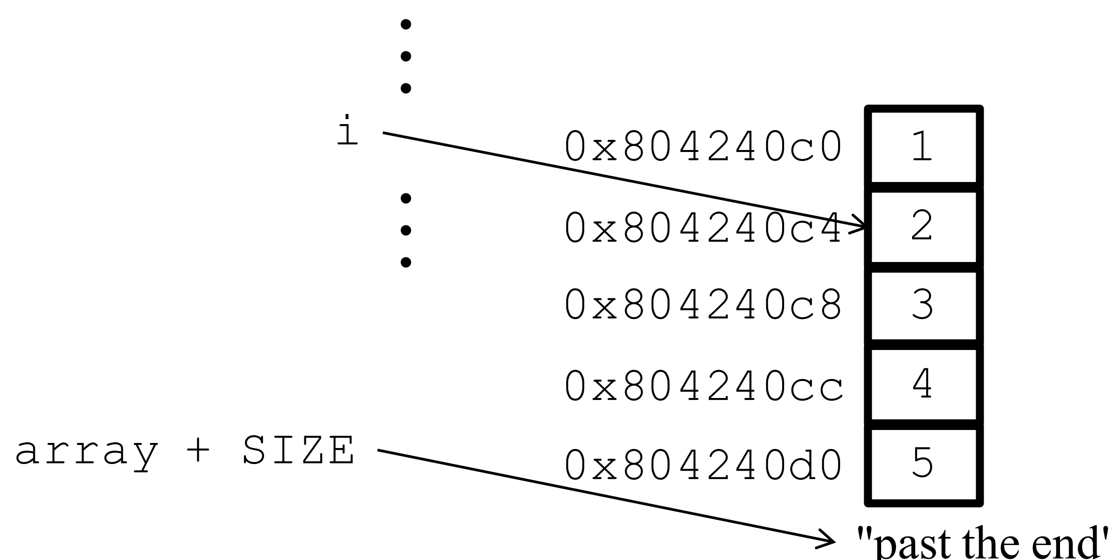
Traversal by pointer

```
int const SIZE = 5;
int array[SIZE] = {1, 2, 3, 4, 5};
for(int *i=array; i < array + SIZE; ++i){
  cout << *i << endl;
}
```



Traversal by pointer

```
int const SIZE = 5;
int array[SIZE] = {1, 2, 3, 4, 5};
for(int *i=array; i < array + SIZE; ++i){
  cout << *i << endl;
}
```



Why arrays?

- Efficiency: arrays are a low level abstraction of memory you can use to write blazing fast code!
  - But can’t I just use a `std::vector`?
    It’s just as fast and easier to use in practice
  - Good point.
    But there's no `std::vector` if you're writing C!

- Learning: arrays are a low level abstraction of memory that gives insight into…
  - …working directly with memory
  - …the pros/cons of contiguously allocated containers
  - …how containers like `std::vector` work under the hood

Where does an array end?

- What happens if a pointer wanders outside of its array?

```
int array[5] = {1,2,3,4,5};
cout << array[42];
```

  - Undefined behavior!
  - You end up reading/writing random memory
  - Program might crash, or maybe not
    Or maybe only sometimes ← Horrifying!
  - Anything can happen, *including getting the right answer*

- How do we keep pointers inside their arrays?
  - Keep track of the length separately
  - **Put a sentinel value at the end of the array**

C-style strings

- In the old days of the C language, strings were originally represented as just an array of characters

```
char str[6] = {'h','e','l','l','o','\0'};
```

- Notice that `str` has 6 chars, not 5
- There is a **null character** at the end of every string

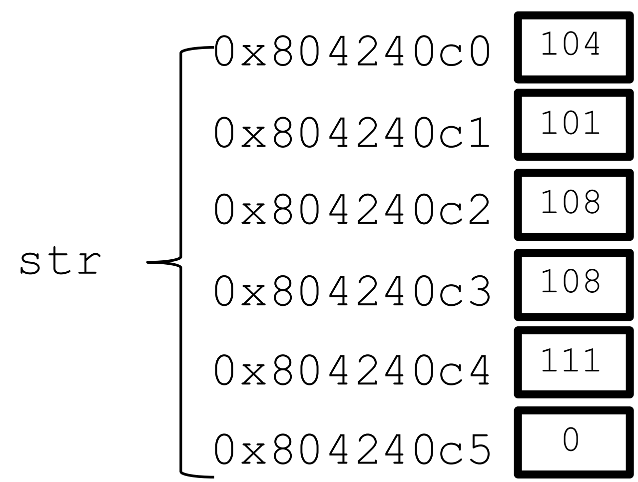
  ```
  char str[6] = {'h','e','l','l','o','\0'};
  ```

  - `'\0'` in code
  - Acts as a **sentinel** to say "Whoa, the array stops here!"

C-style strings

- There is a **null character** at the end of every string
  `char str[6] = {'h','e','l','l','o','\0'};`
  - `'\0'` in code
  - Acts as a **sentinel** to say “Whoa, the array stops here!”

- In memory, this looks like:

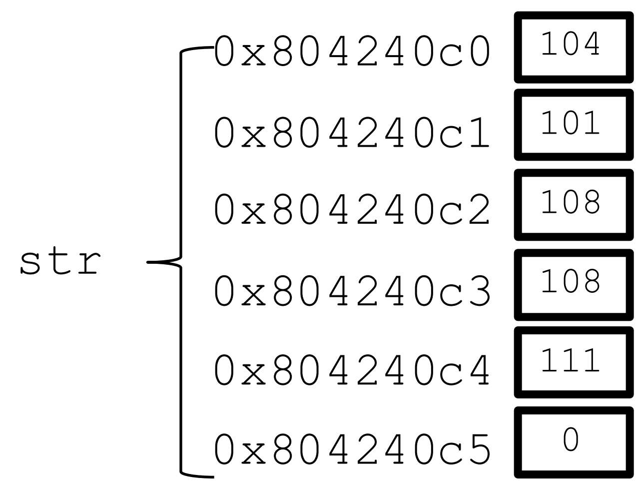


C-style strings

- Why are the memory locations filled with numbers?
- `char` objects are really numbers under the hood (ASCII)\
- `char` objects occupy one byte
    - `sizeof(char) == 1`

AKA `NULL`
AKA `false`

Symbol	Number
'\0'	0
...	
'e'	101
'f'	102
'g'	103
'h'	104
...	



C-style string

```
char str[6] = "hello";
```

- Short cut for initializing strings:

```
char str[] = "hello";
```

Compiler automatically puts `'\0'` at the end of string literals

- Of course, these turn into pointers as well

```
char *str_ptr = str;
```

C-string pitfalls

- What does this code *actually* do?

```
char str1[6] = "hello";
char str2[6] = "hello";
char str3[6] = "apple";
char *ptr = str1;

// Test for equality?
str1 == str2;

// Copy strings?
str1 = str3;

// Copy through pointer?
ptr = str3;
```

Traversing a C-string

- Just keep going until we find the **sentinel**
  - When the current element has value `'\0'`

```
char str[6] = "hello";
int myfunction(char *str){
  char *ptr = str;
  while(*ptr != '\0'){
    ++ptr;
  }
  return ptr - str;
}
```

Pointer starts at beginning of the array

Continue until **pointer** at end

Increment **pointer**

- What does this function do? Draw a picture of memory.

The const Keyword

- This function should never modify the original string
- Let's get the compiler to help catch mistakes

```
int strlen(const char *str){
  const char *ptr = str;
  while(*ptr != '\0'){
    ++ptr;
  }
  return ptr - str;
}
```

- `const` is a *type qualifier* – something that modifies a type
- It means “you cannot change this value once you have initialized it”

The const Keyword

- When you have pointers, there are two things you might change:
    1. The value of the pointer
    2. The value of the object to which the pointer points

- Either (or both) can be made unchangeable:

```
const T *p;  // "T" (the pointed-to object)
             // cannot be changed
T *const p;  // "p" (the pointer) cannot be
             // changed
const T *const p; // neither can be changed
```

The const Keyword

- You can use a pointer-to-T anywhere you expect a pointer-to-const-T, but NOT vice versa

```
int strlen(const char *str);
void scramble(char *str);
const char *s1 = "can't change me";
char s2[] = "go for it";
strlen(s1);
strlen(s2);
scramble(s1);
scramble(s2);
```

- Which lines cause a compiler error?

Pointer Exercise: Code these

```
//REQUIRES: "a" points to an array of length "size"
//EFFECTS:  Returns a pointer to the first
//  occurrence of "search" in "a".
//  Returns NULL if not found.
int * find (int *a, int size, int search);

//REQUIRES: "s" is a NULL-terminated C-string
//EFFECTS:  Returns a pointer to the first
//  occurrence of "search" in "s".
//  Returns NULL if not found.
char * strchr (char *s, char search);
```

Do not use array indexing, e.g., `a[i]` or `*(a+i)`