Slides by Andrew DeOrio
and James Juett

EECS 280

Programming and Introductory Data Structures

Pointers

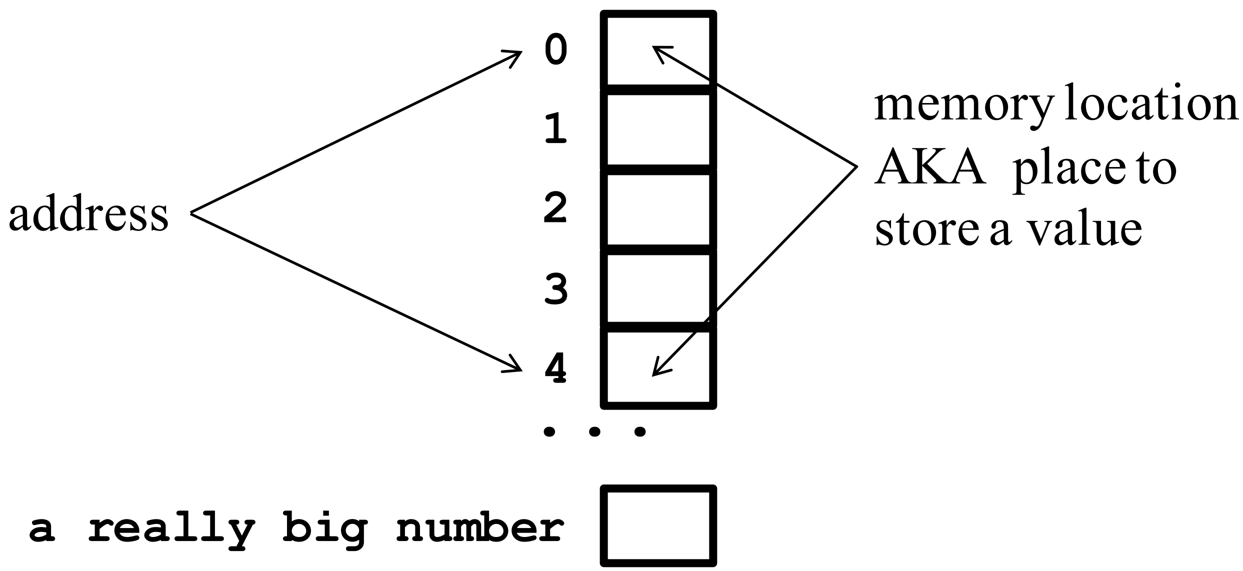
C++ review

- An **object** is a chunk of memory that holds some **value** from the possible set of values for the object's **type**
- A **variable** is a name that refers to an object

- For example
  ```
  int x = 3;
  ```
  - The **value** is `3`
  - The **type** is `int`
  - The **variable** is `x`
- The name `x` refers to the new object

Thinking about 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**



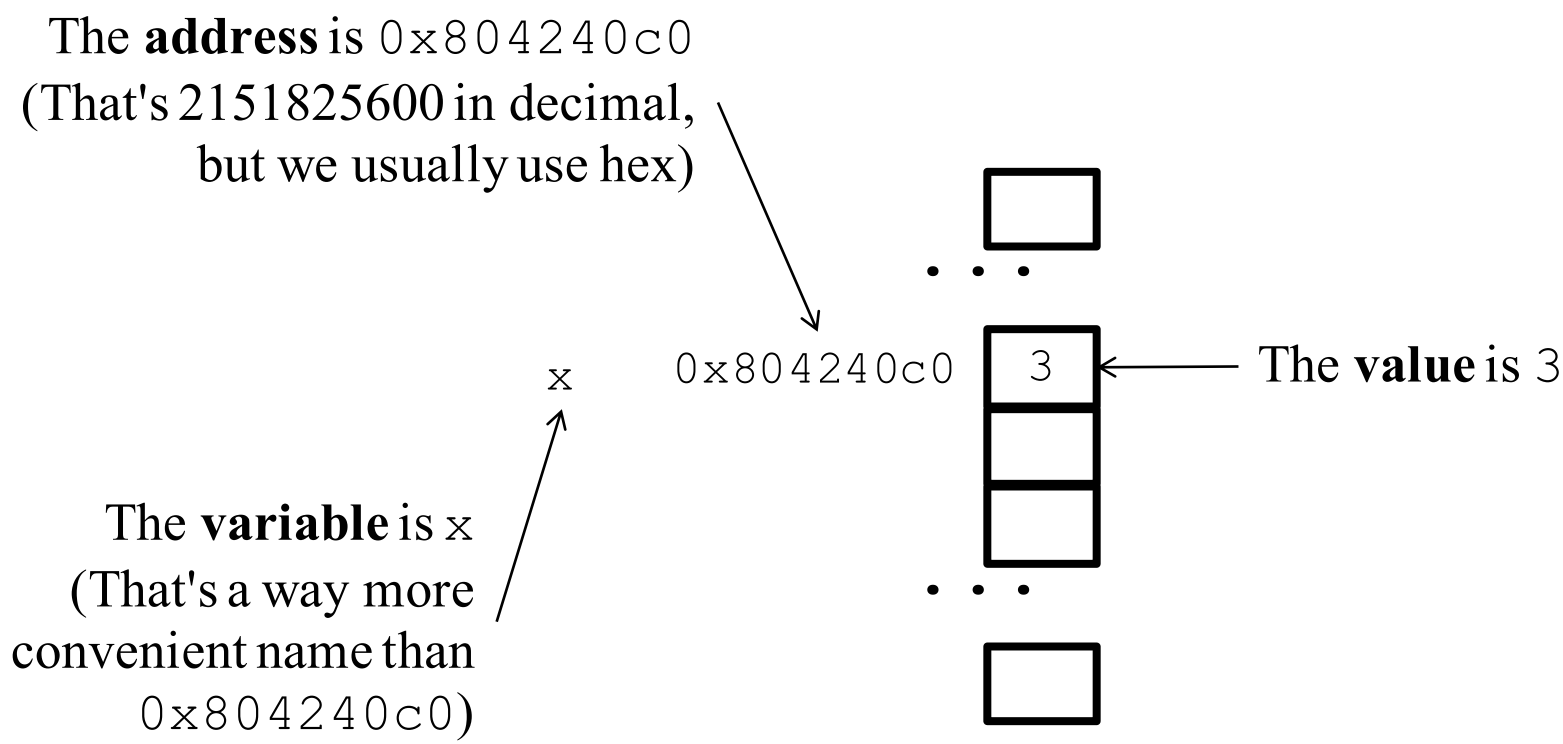
Numbers in binary and hex

- Addresses are often expressed in **hexadecimal**
- "hex" for short
- 42 in base 10 (AKA decimal)
  $= 40 + 2$
  $= 4*10^1 + 2*10^0$
  $= 42_{10}$
- 42 in base 2 (AKA binary)
  $= 32 \quad + \quad 0 \quad + \quad 8 \quad + \quad 0 \quad + \quad 2 \quad + \quad 0$
  $= 1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$
  $= 101010_2$
- 42 in base 16 (AKA hexadecimal)
  $= 32 \quad + \quad 10$
  $= 2*16^1 + 10*16^0$
  $= 2A_{16}$
  or
  $= 0010\ 1010_2$
  $= \quad 2 \qquad A$
  $= 0x2A$

Decimal	Binary	Hex
0	0000	0x0
1	0001	0x1
2	0010	0x2
3	0011	0x3
4	0100	0x4
5	0101	0x5
6	0110	0x6
7	0111	0x7
8	1000	0x8
9	1001	0x9
10	1010	0xA
11	1011	0xB
12	1100	0xC
13	1101	0xD
14	1110	0xE
15	1111	0xF

Thinking about memory

```
int x = 3;
```



C++ review

- C++ uses **value semantics**, which means initialization and assignment involve **copying** the **value** from one object to another

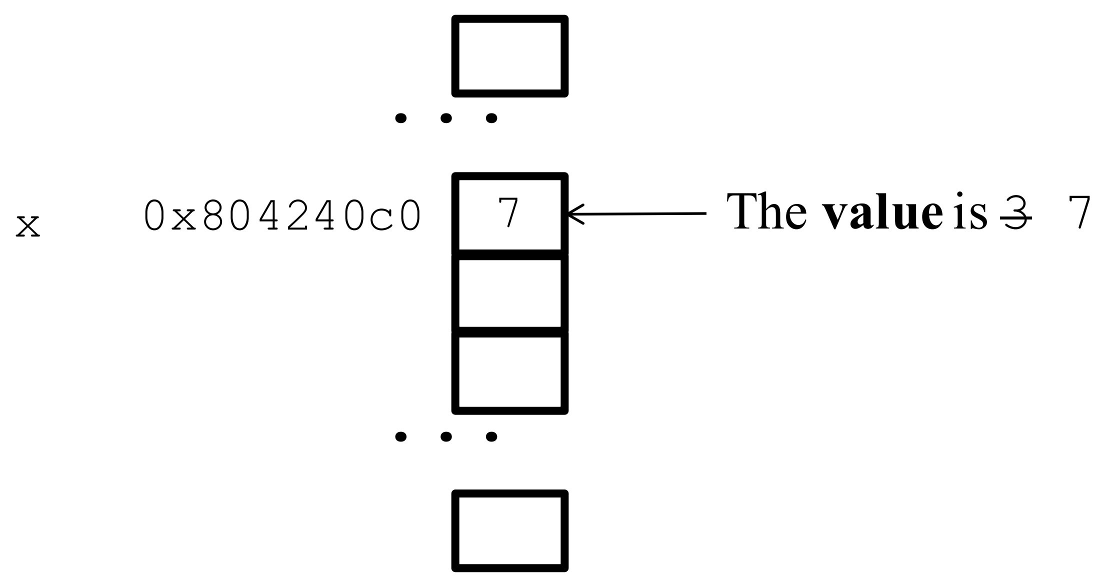
- **Initialization**
  - Giving an object an initial value when it is created
  - Parameter passing works like initialization
    ```
    int x = 3;
    int x(3);
    ```

- **Assignment**
  - Overwrite old value of an object with new value
  - Takes place in context of an expression
    ```
    x = 3;
    ```

Thinking about memory

- Assignment overwrites the **value** in a **memory** location

```
int x = 3;
x = 7;
```

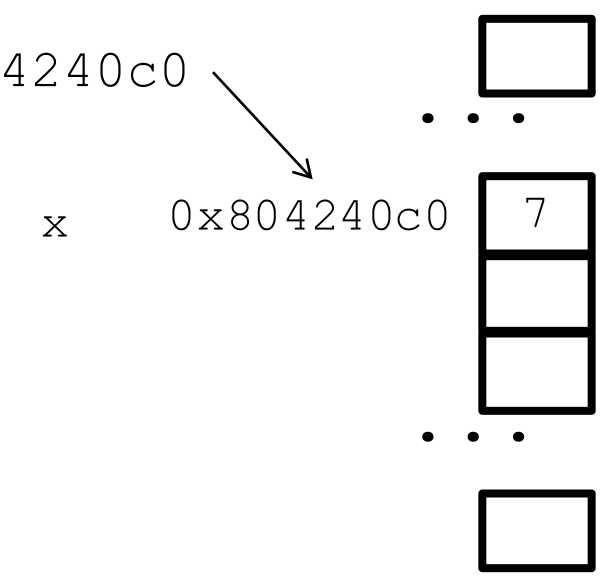


Thinking about memory

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

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

The **address** is `0x804240c0`



Address-of vs. reference type

- Don't confuse the *address of* operator with a *reference type*

- `int x;`
  `x` is a variable whose type is `int`

- `int &r;`
  `r` is a variable whose type is `reference-to-int`

- `int swap(int &a, int &b);`
  `a` and `b` are variables whose type is `reference-to-int`

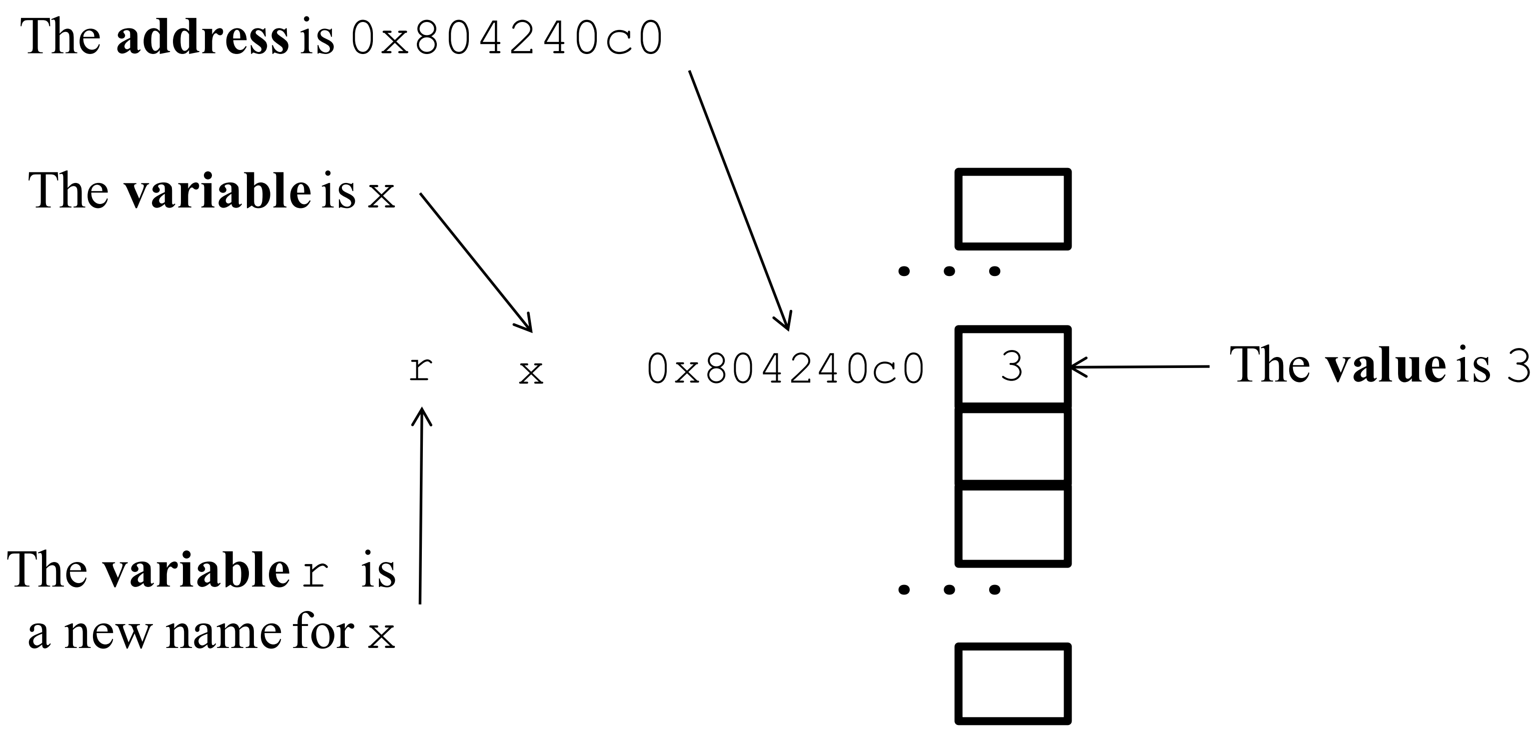
Address-of vs. reference type

- Recall that a **reference** is a new name for an existing object

- For example:
```
int x = 3;
int &r = x;
```
- The **type** is `reference-to-int`
- The **variable** is `r`
- This code does not create any new objects

Address-of vs. reference type

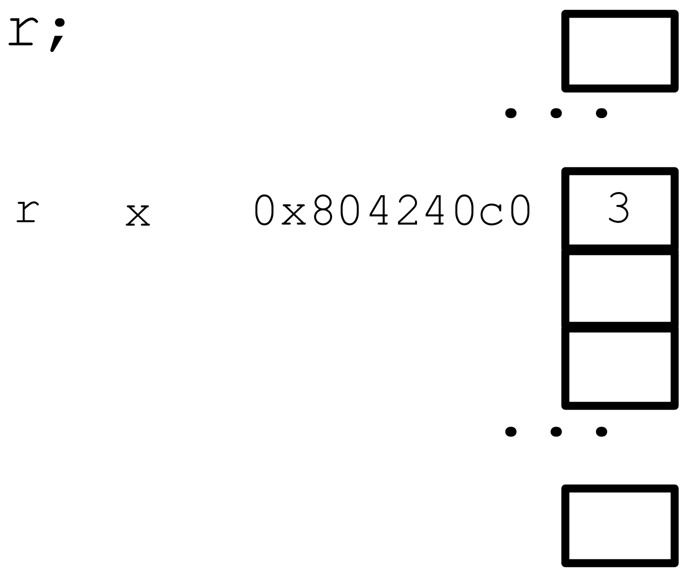
```
int x = 3;
int &r = x;
```



Address-of vs. reference type

```
int x = 3;
int &r = x;
cout << x;
cout << r;
cout << &x;
cout << &r;
```

What is the output of this code?



Pointers

- What is the type of the object returned by the address-of operator?

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

- Answer: pointer

Pointers

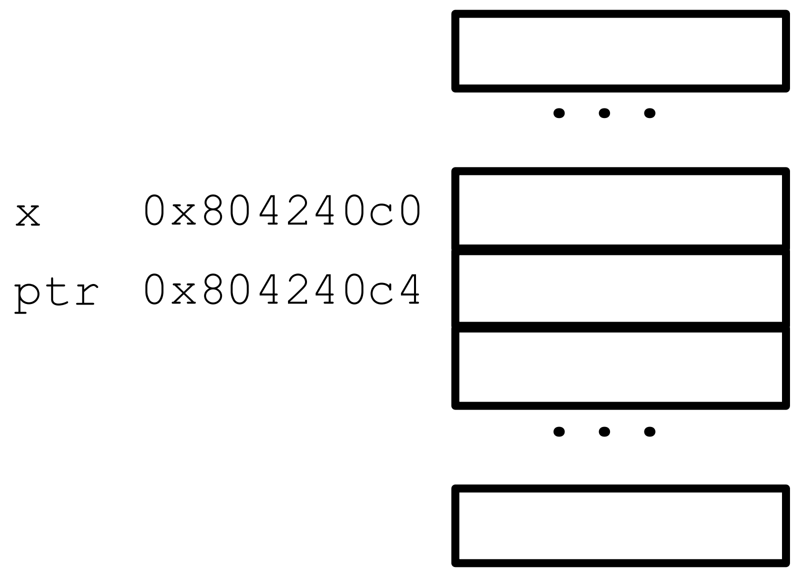
```
int x = 7;
cout << &x; //0x804240c0
____ ptr = &x;
```

- A pointer is a type of object whose **value** is the **address** of an object
- To declare a pointer variable, affix a `*` to the left of the name
  ```
  int *ptr = //...
  ```
- There is a separate pointer type for each kind of thing you could point to, and you can't mix them

Thinking about memory

```
int x = 7;
int *ptr = &x;
```

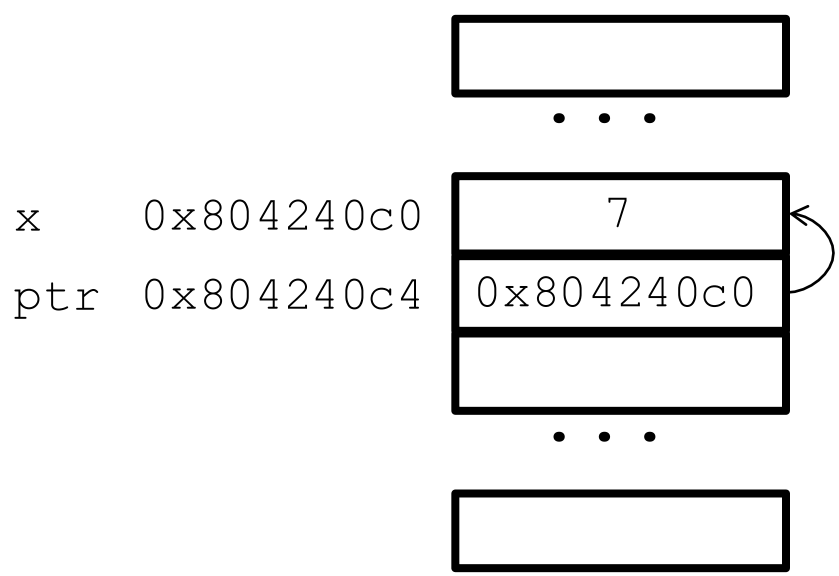
Exercise: fill in the values



Thinking about memory

```
int x = 7;
int *ptr = &x;
```

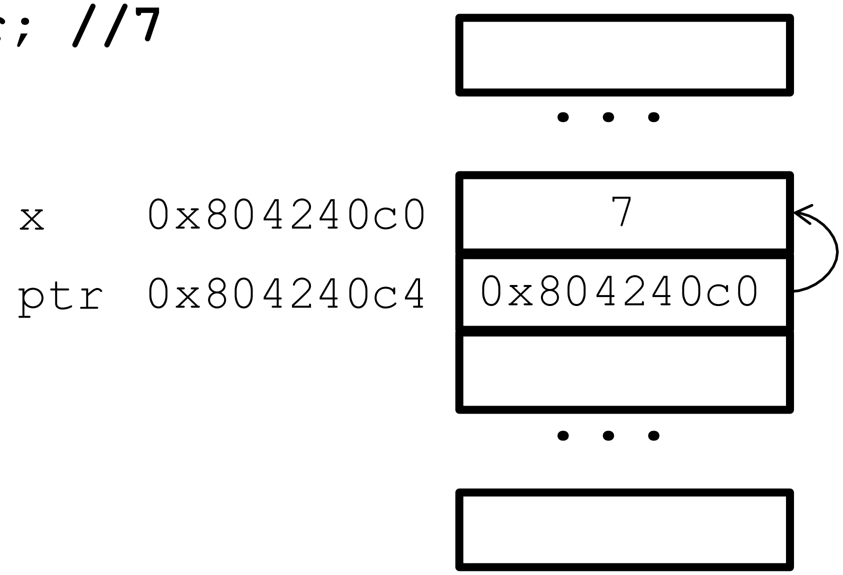
We say that `ptr` "points to" `x`



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
```



Exercise

- What is the output?

```
int foo = 1;
int *bar = &foo;
foo = 2;
*bar = 3;

cout << foo;
cout << bar;
cout << *bar;
```

Exercise

```
int *x, *y;
int a = -1;

x = &a;
cout << *x;
*x = 42;
cout << a;
*y = 13;
cout << *y;
y = x;
cout << *y;
cout << a;
```

What can you do with pointers?

- Pointers let us work with objects *indirectly*
  - Similar to reference semantics
  - Use objects across different scopes
  - Keep track of objects in dynamic memory[1]

1 We'll look at this when we get to dynamic memory

Working with objects indirectly

```
void add_one(int *x){
 *x += 1; // works with object b, even though the
          // name b is not in scope here
}

int main(){
  int a = 1, b = 2;
  int *ptr = &a; //*ptr "points to" a
  ptr = &b;      //now *ptr "points to" b
  add_one(ptr);  //adds one to b (pointed by ptr)
  add_one(&b);   //adds one to b (pointed by ptr)
}
```

Working with objects indirectly

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

int main(){
  int a = 1, b = 2;
  int *ptr = &a; //*ptr "points to" a
  ptr = &b;      //now *ptr "points to" b
  add_one(ptr);  //adds one to b (pointed by ptr)
  add_one(&b);   //adds one to b (pointed by ptr)
}
```

Before pointers, we had to know the name of an object to use it, and the name had to be in scope

We actually changed which object `ptr` "points to". This is reference semantics!

What can you do with pointers?

- Pointers let us work with objects *indirectly*
  - Similar to reference semantics
  - Use objects across different scopes
  - Keep track of objects in dynamic memory[1]

- Work with *arrays* of objects
  - Objects in arrays have *sequential addresses*
  - We can do *pointer arithmetic* to compute the address of the element we want

1 We'll look at this when we get to dynamic memory

Kinds of objects in C++

- **Atomic**
  - Also known as **primitive**
  - `int, double, char,` etc.
  - Pointer types

- **Arrays** (homogeneous)
  - A *contiguous* sequence of objects of the same type

- **Class-type** (heterogeneous)
  - A compound object made up of member sub-objects
  - The members and their types are defined by a **class**

Arrays in C++

- In C++ an array is a very simple *collection* of objects

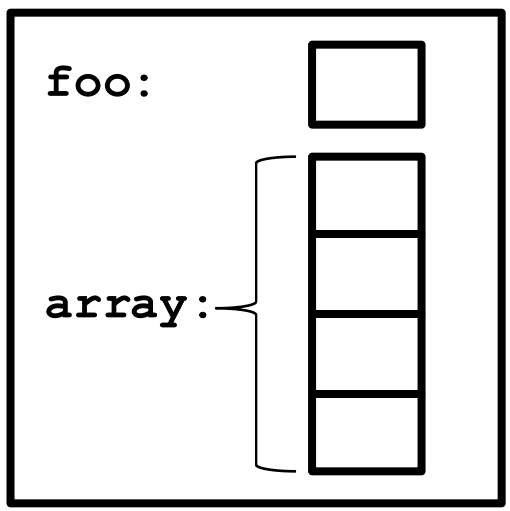
- Arrays…
  - …have a fixed size
  - …hold elements of all the same type
  - …have ordered elements
  - …occupy a *contiguous* chunk of memory
  - …support constant time random access (i.e. “indexing”)

Defining arrays

- For comparison purposes, let's also declare and define an integer, foo:

```
int foo;
int array[4];
```

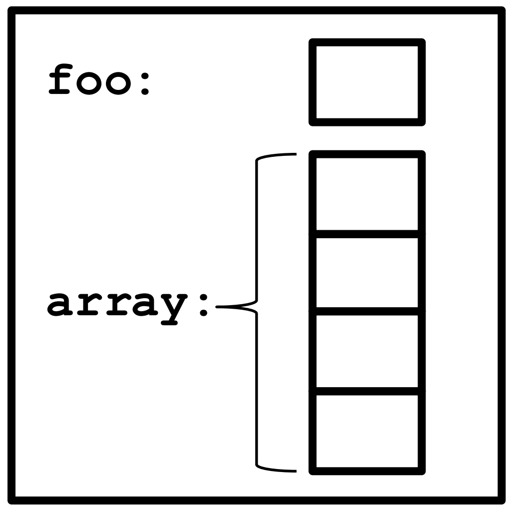
- The environment that we get when we do this is:



Defining arrays

```
int foo;
int array[4];
```

- What are the contents of “array” after this declaration?



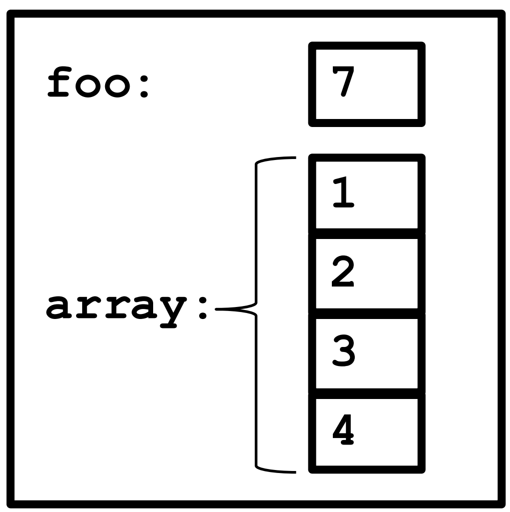
Initializing arrays

- You can also initialize the contents of an array in one line – just like with an int. However, we need some sort of notation to specify a set of numbers:

This is called a “static initializer”.

```
int foo = 7;
int array[4] = { 1, 2, 3, 4 };
```

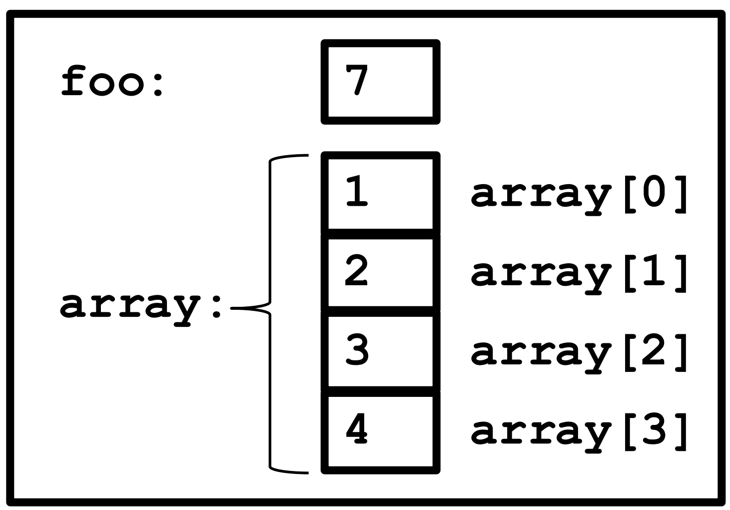
- The corresponding environment would look like this:



Accessing array elements

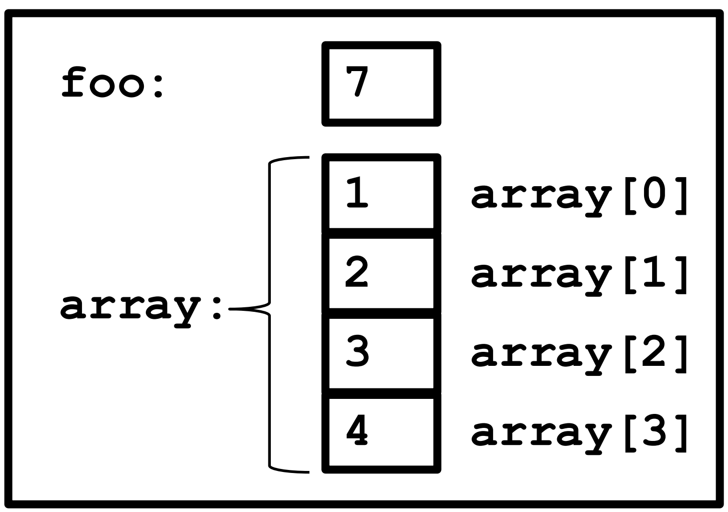
- You can access the contents of an array using an “index”. The index of the first array element is zero, the next is one, and so on.

- So, we can name the individual elements of array, like so:



Accessing array elements

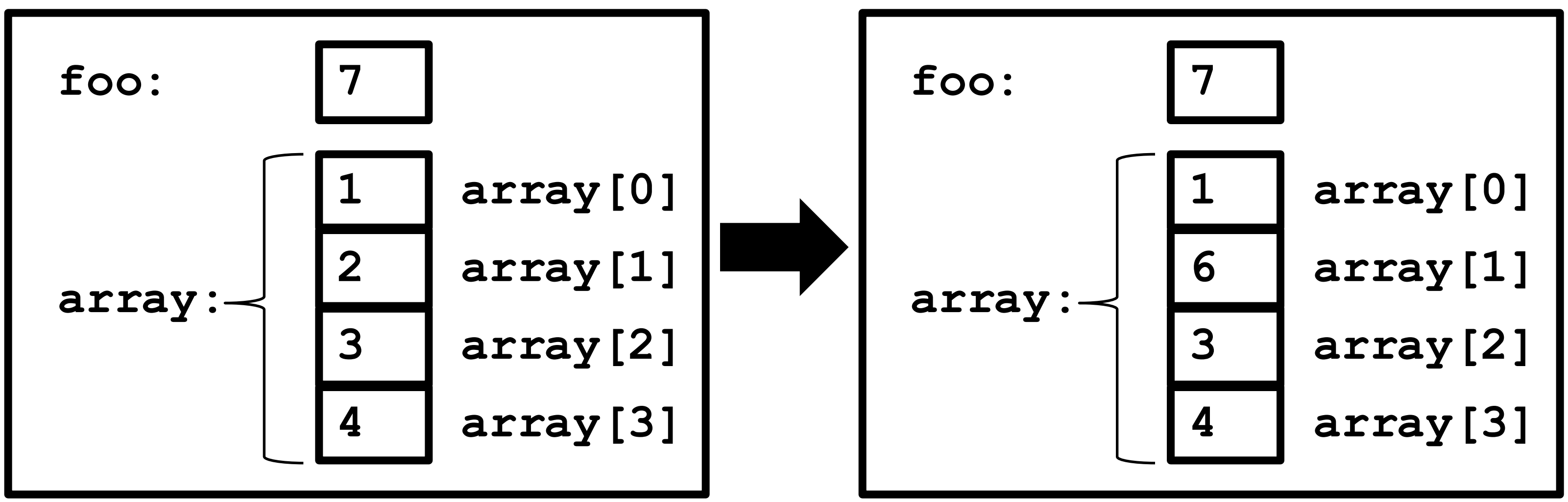
- Each individual element is used just like a regular int, so all of the following are legal:



Accessing array elements

- Each individual element is used just like a regular int, so all of the following are legal:

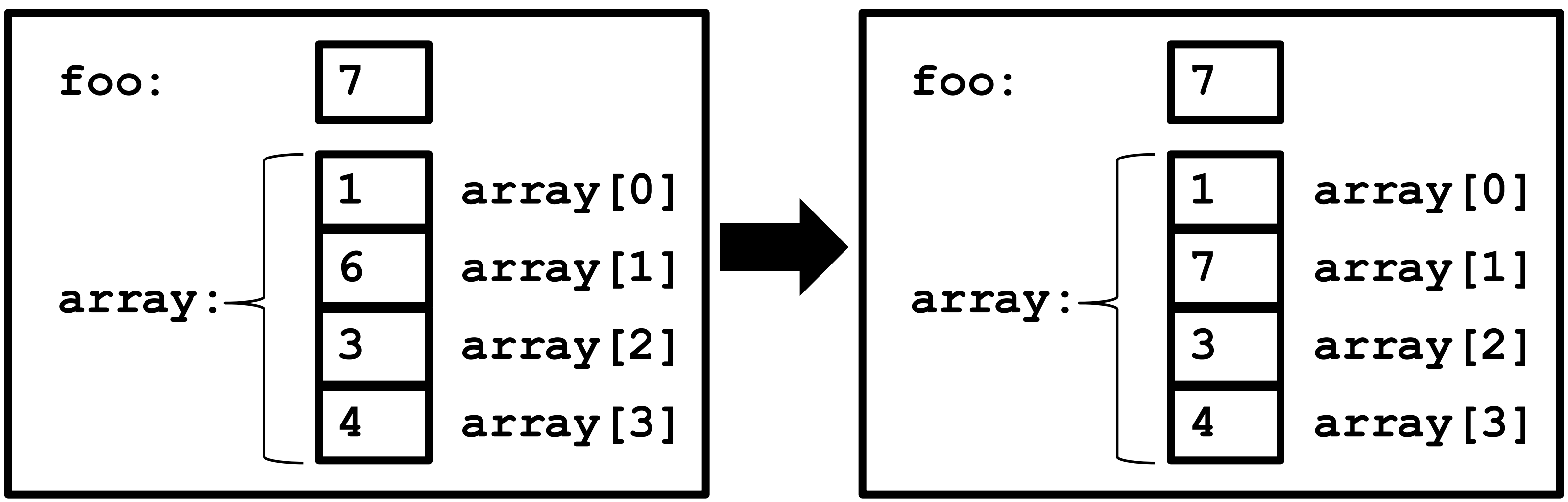
```
array[1] = 6;
```



Accessing array elements

- Each individual element is used just like a regular int, so all of the following are legal:

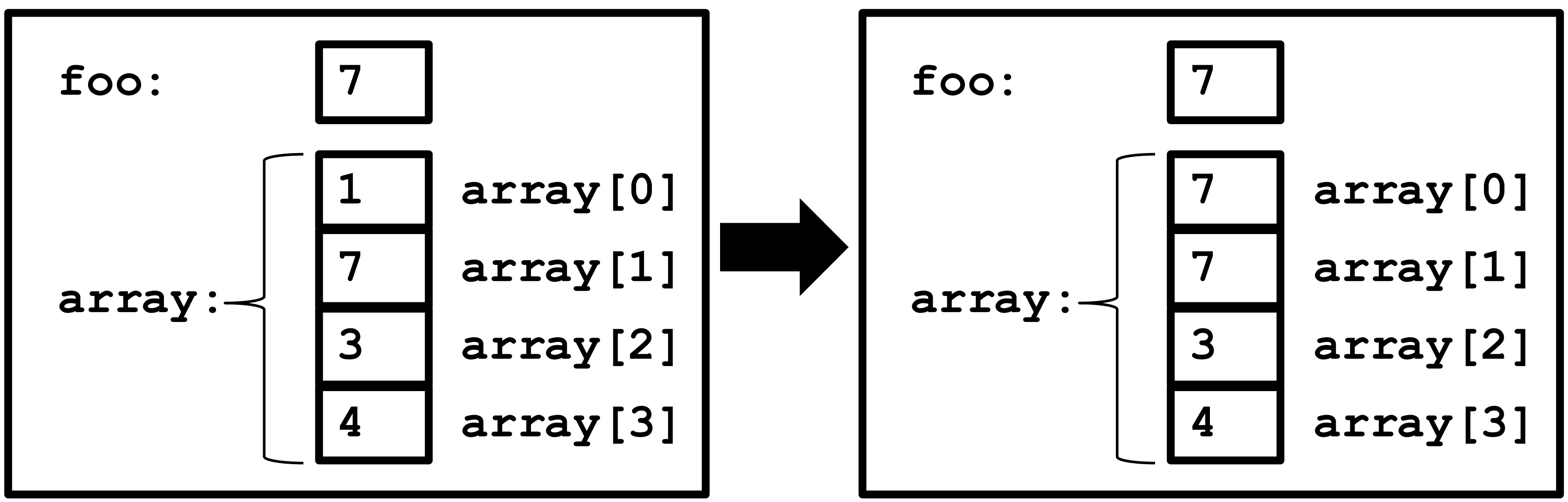
```
array[1] = 6;
++array[1];
```



Accessing array elements

- Each individual element is used just like a regular int, so all of the following are legal:

```
array[1] = 6;
++array[1];
array[0] = array[1];
```

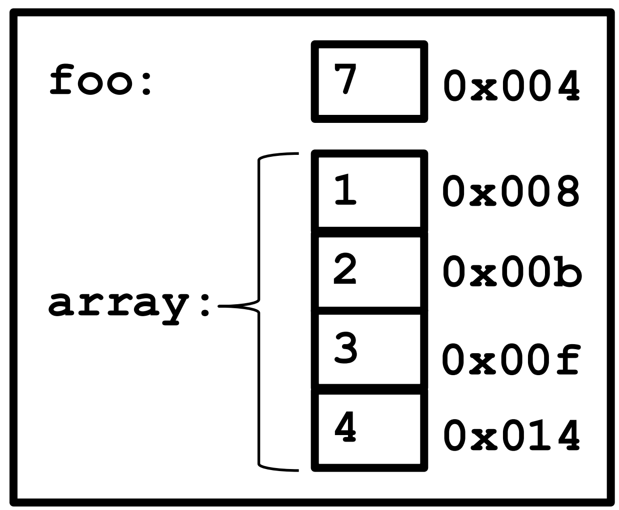


The dark secret of arrays

- In C++, arrays are **objects** with no **value**
  - The individual elements have values, of course, but not the array as a whole!

- Try to get the value of an array…
  - It suddenly turns into **a pointer to its first element**

```
int foo = 7;
int array[4] = {1,2,3,4};
cout << array; //0x008
```



The dark secret of arrays

- The tendency of arrays to turn into pointers has a lot of consequences

- You can't assign arrays to each other

```
int array1[4] = {1,2,3,4};
int array2[4] = {5,6,7,8};
array2 = array1;
```

Still an array

Already turned into a pointer

The dark secret of arrays

- The tendency of arrays to turn into pointers has a lot of consequences

- Array parameters (pass by value)

Compiler changes to:
`int *array`

```
int array_max(int array[4]){
  // find the max elem and return it
}

int array[4] = {1,2,3,4};
array_max(array)
```

Turns into a pointer before being passed

Array indexing revisited

- How does array indexing work?

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

Turns into a pointer before the `[2]` part

- Array indexing actually does *pointer arithmetic* followed by a *dereference*
- `array[i]` is the same thing as `*(array + i)`
- When you add an integer to a pointer, it computes the address offset by some number of objects according to the pointer type

Don't do this. Ever.

- We know this equivalence:

```
array[i] = *(array+i)
```

- Let’s try something…

```
array[i]
*(array+i)
*(i+array)
i[array]
```

- Yeah, that actually works
- Never do this again

Exercise

- Pointers don't know anything about how big an array is
- Because arrays convert to pointers, a function won't know how big an array input is
- We include a size to fix this

```
int array_max(int array[], int size) {

}

int main() {
  int array[4] = {1,2,3,4};
  array_max(array)
}
```

Exercise

- Functions passing arrays are usually written with pointer syntax

```
int array_max(int *array, int size) {


}


int main() {
  int array[4] = {1,2,3,4};
  array_max(array)
}
```

Array size

- What's wrong with this code?

```
int array_max(int *array, int size) {
  assert(size > 0);
  int m = array[0];
  for (int i=1; i <= size; ++i) {
    if (m > array[i]) m = array[i];
  }
  return m;
}

int main() {
  int array[4] = {1,2,3,4};
  array_max(array, 4)
}
```

Array size

- What's wrong with this code?

```
int array_max(int *array, int size) {
  assert(size > 0);
  int m = array[0];
  for (int i=1; i <= size; ++i) {
    if (m > array[i]) m = array[i];
  }
  return m;
}
```

- The compiler *cannot check* for going off the end of an array!
- Why? Because it's a pointer!
  - Note: you'd have the same problem with `int array[]` synatx