Pointers and Arrays

Basic Variables

Variables store data of a specific type

```
int num;        // a variable that can store an integer
num = 42;       // like 42
```

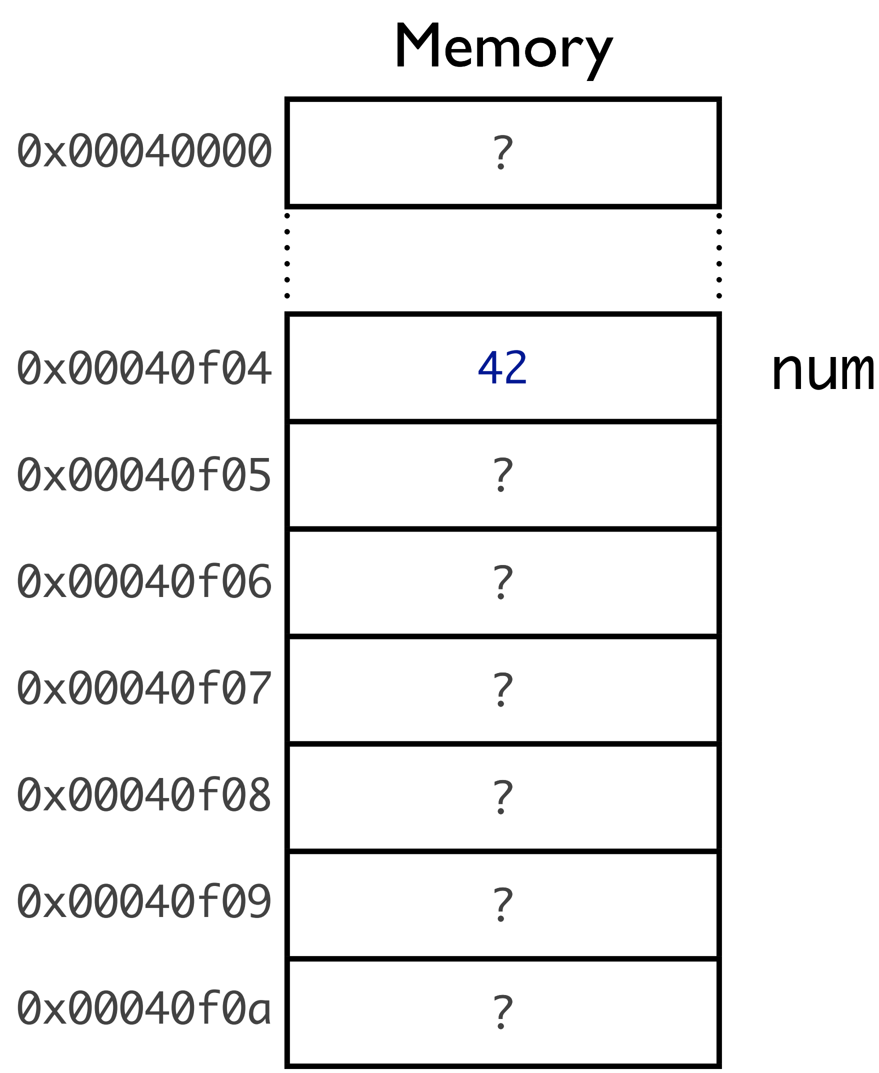
Think of them as little boxes that can only store certain types of data:

`int num` $\boxed{42}$

Basic Variables

In reality, a variable is stored at a specific address in memory

```
int num = 42;  // stored at address 0x00040f04
```



Address Operator (&)

In reality, a variable is stored at a specific address in memory

```
int num = 42;  // stored at address 0x00040f04
```

We can ask for that address using the address operator (&):

```
cout <<  num << end; // displays: 42

cout << &num << end; // displays: 0x00040f04
```

`the address operator`

A variable can be referred to by *either* its name or its address

- this proves extremely useful, as we'll see...

Pointers

What if we wanted to store the address into a variable?

- what type of variable would we use?
- it depends on the data type that lives at that address...

To store the address of an int, use a pointer to an int:

```
int* ptr;     // a pointer to an int, called ptr
```

use * to indicate a pointer

Pointers

What if we wanted to store the address into a variable?

- what type of variable would we use?
- it depends on the data type that lives at that address...

To store the address of a double, use a pointer to a double:

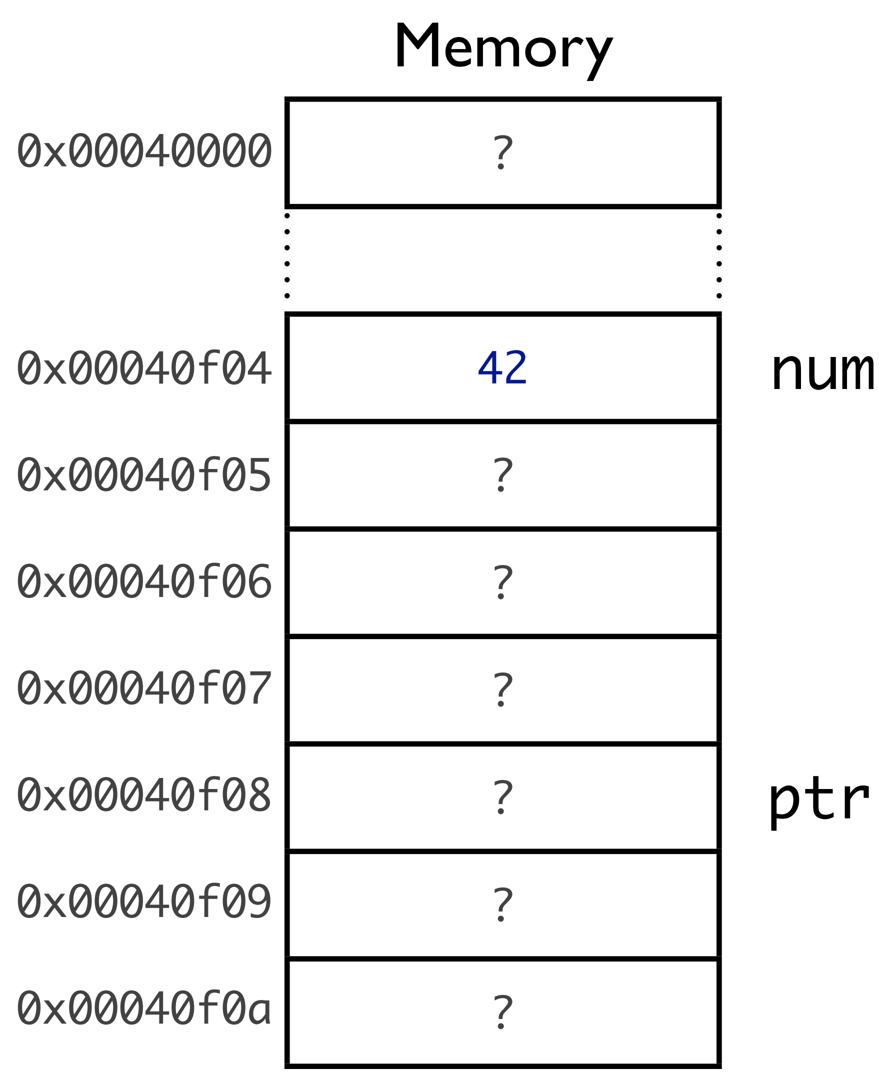
```
double* orc;  // a pointer to a double, called orc
```

use * to indicate a pointer

Pointers

Pointers are simply variables that store memory addresses

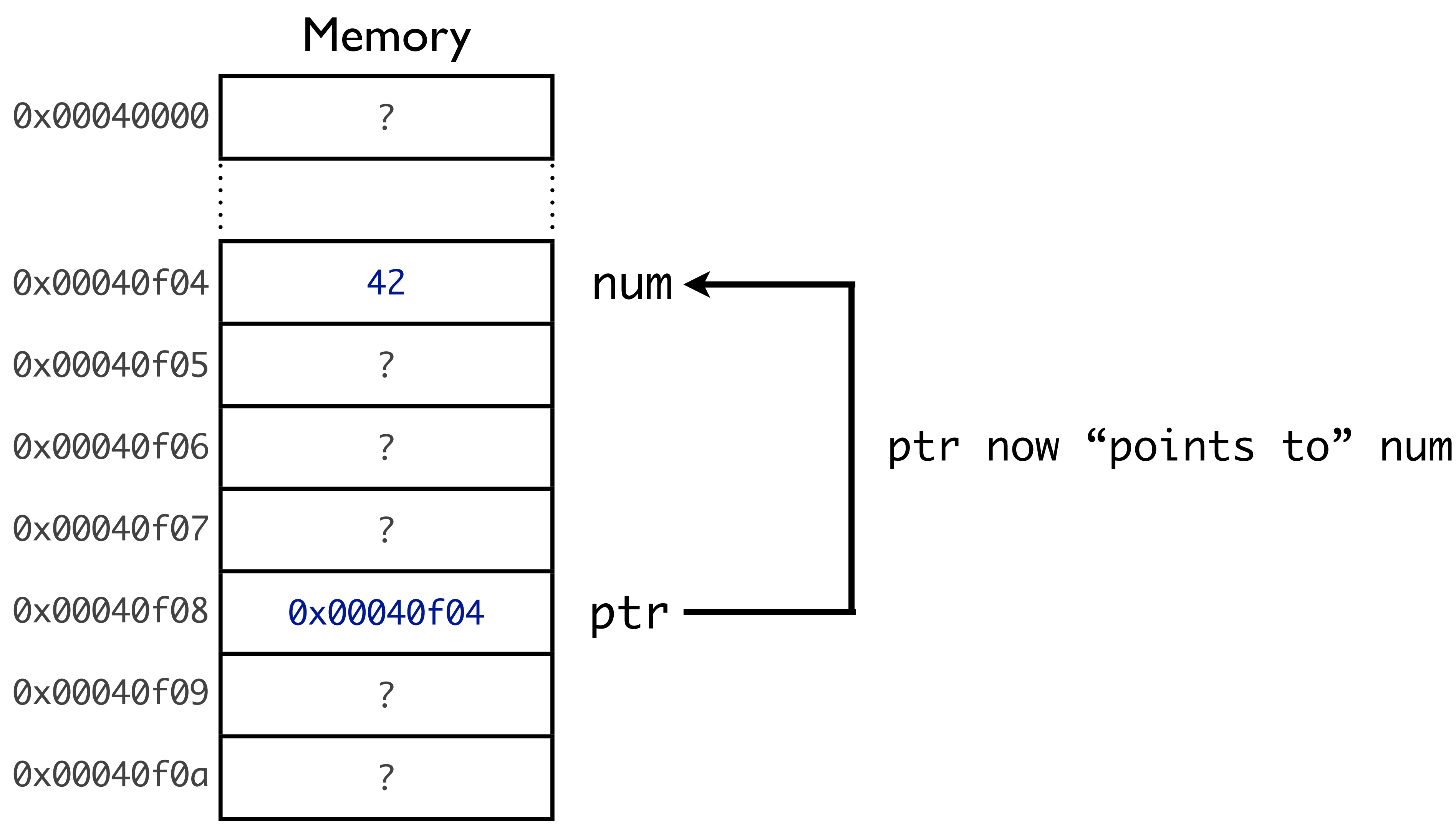
```
int num = 42;       // stored at address 0x00040f04
int* ptr;           // a pointer to an int
```



Pointers

Pointers are simply variables that store memory addresses

```
int num = 42;      // stored at address 0x00040f04
int* ptr = &num;   // set ptr to the address of num
```

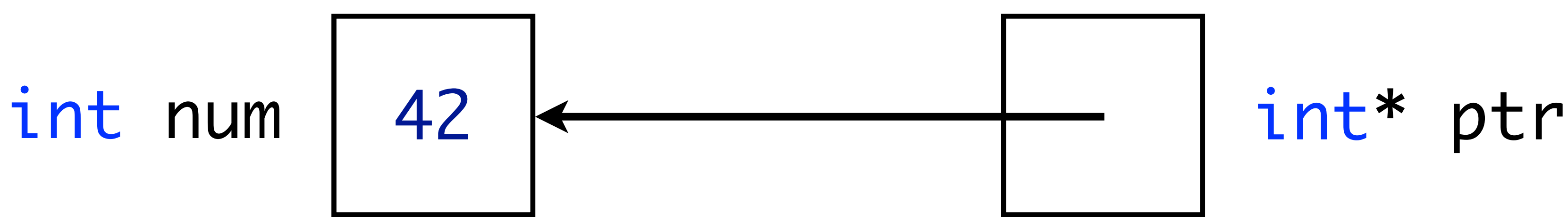


Pointers

Pointers are simply variables that store memory addresses

```
int num = 42;      // stored at address 0x00040f04
int* ptr = &num;   // set ptr to the address of num
```

We say that `ptr` "points to" `num`:



Pointers

Pointers are simply variables that store <u>memory addresses</u>

```
int num = 42;      // stored at address 0x00040f04
int* ptr = &num;   // set ptr to the address of num
```

Printing the values stored in these variables:

```
cout <<  num << endl;  // displays: 42
cout <<  ptr << endl;  // displays: 0x00040f04
cout << *ptr << endl;  // displays: 42
```

this is called “dereferencing” the pointer

Pointers

Think of pointers as a way to alias variables...

```
int BruceWayne;         // the superhero's true identity
int* Batman;            // the superhero's alias
Batman = &BruceWayne;   // links pointer to variable
```

Spoiler alert:

- `BruceWayne` and `*Batman` are really the same person!

==

gasp!

Pointers

Changes to `*Batman` affect `BruceWayne`:

```
*Batman = 42;
cout << *Batman << endl;     // displays 42
cout << BruceWayne << endl; // also displays 42!
```

And vice-versa:

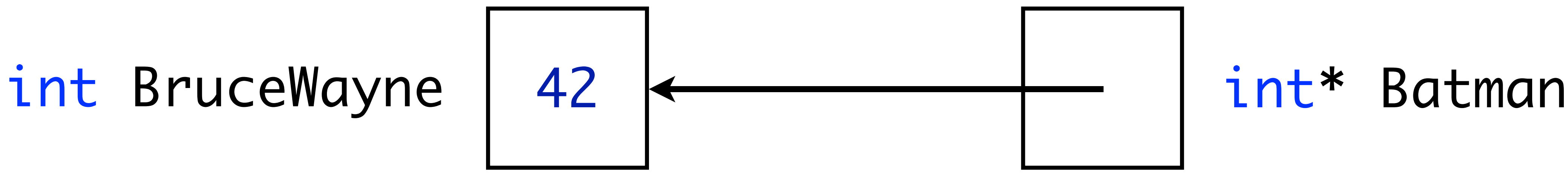
```
BruceWayne = 24;
cout << BruceWayne << endl; // displays 24
cout << *Batman << endl;     // also displays 24!
```

Pointers

Changes to `*Batman` affect `BruceWayne`:

```
*Batman = 42;
cout << *Batman << endl;     // displays 42
cout << BruceWayne << endl; // also displays 42!
```

However, only the value of `BruceWayne` is changing:



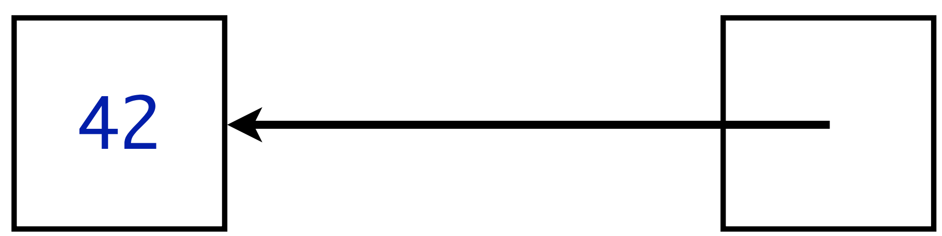
Pointers

Changes to `*Batman` affect `BruceWayne`:

```
*Batman = 42;
cout << *Batman << endl;     // displays 42
cout << BruceWayne << endl; // also displays 42!
```

`Batman` just holds the <u>address</u> of `BruceWayne`...

`int BruceWayne`
`(aka *Batman)`



`int* Batman`

`*Batman` (with the `*`) is going to where the address points (`BruceWayne`)

Pointer Example

Let's say we have these variables:

```
int num = 42;
int *p1, *p2; // both need *'s when declared like this
```

Then we do some assignment:

```
p1 = &num;
p2 = p1;
*p2 = 10;
```

What does this statement output?

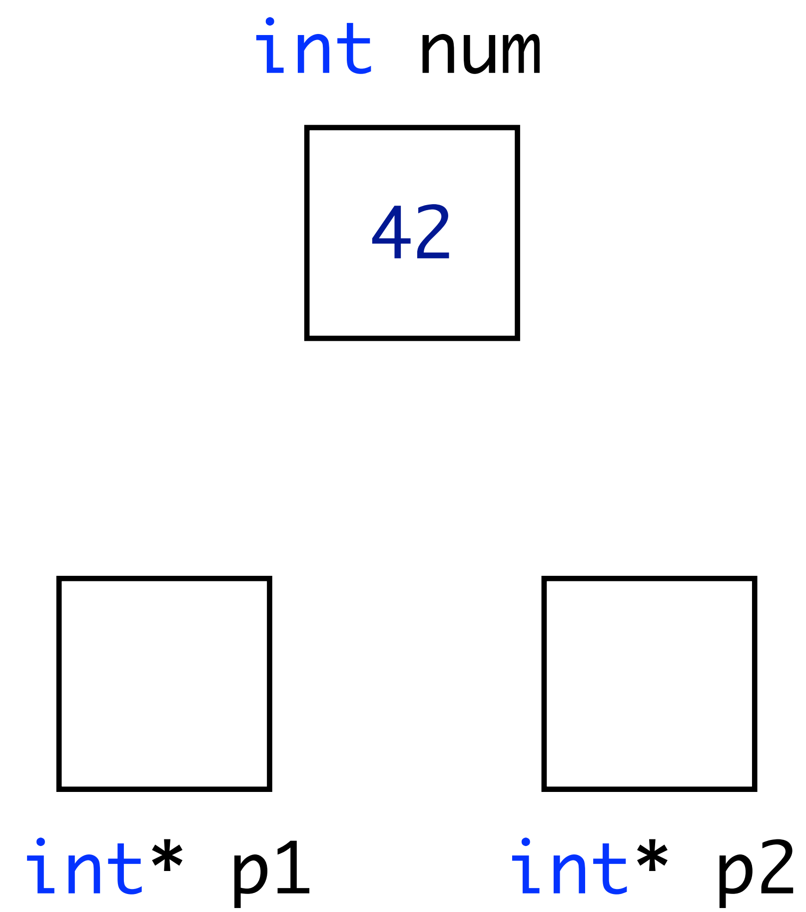
```
cout << num << " " << *p1 << " " << *p2 << endl;
```

Pointer Example

Let's say we have these variables:

```
int num = 42;

int *p1, *p2;
```



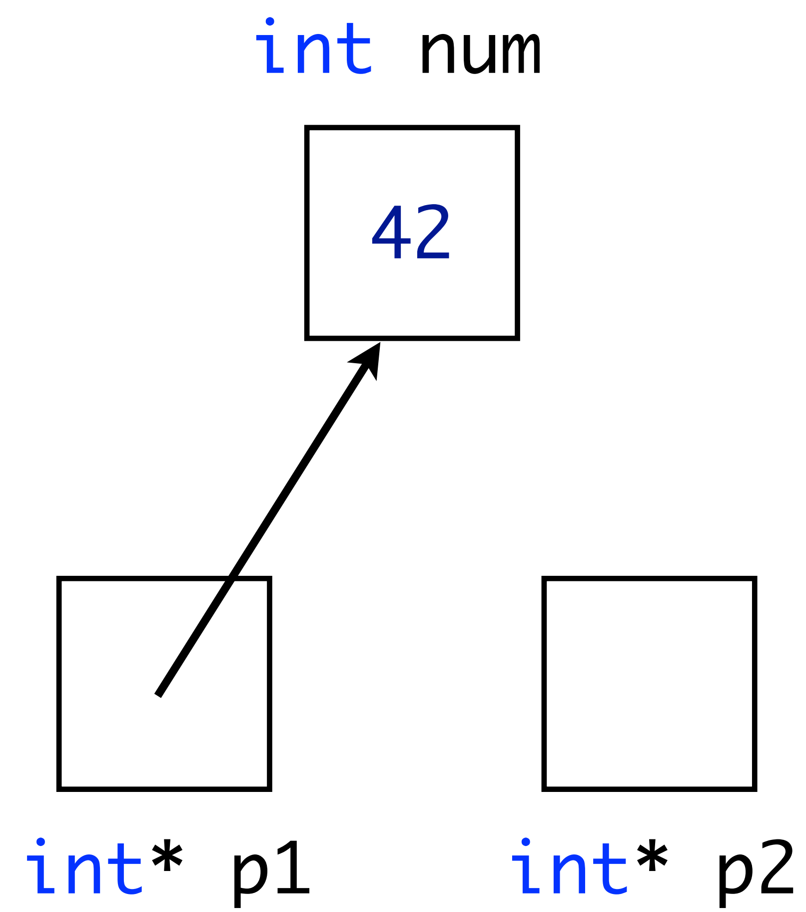
Pointer Example

Let's say we have these variables:

```
int num = 42;
int *p1, *p2;
```

Then we do some assignment:

```
p1 = &num;
```



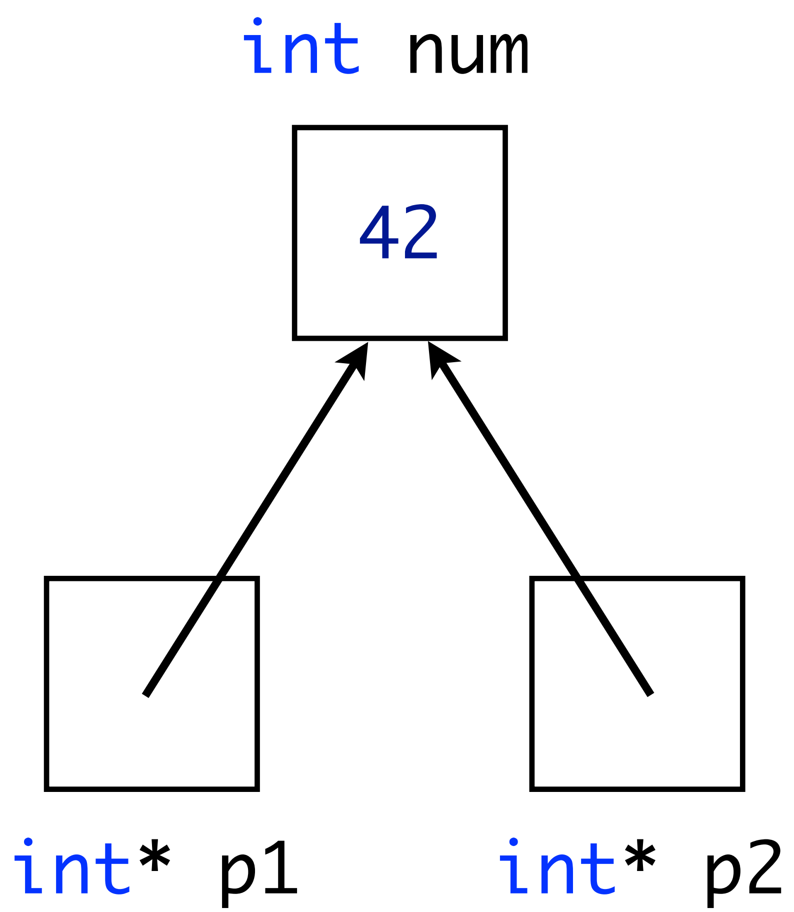
Pointer Example

Let's say we have these variables:

```
int num = 42;
int *p1, *p2;
```

Then we do some assignment:

```
p1 = &num;
p2 = p1;
```



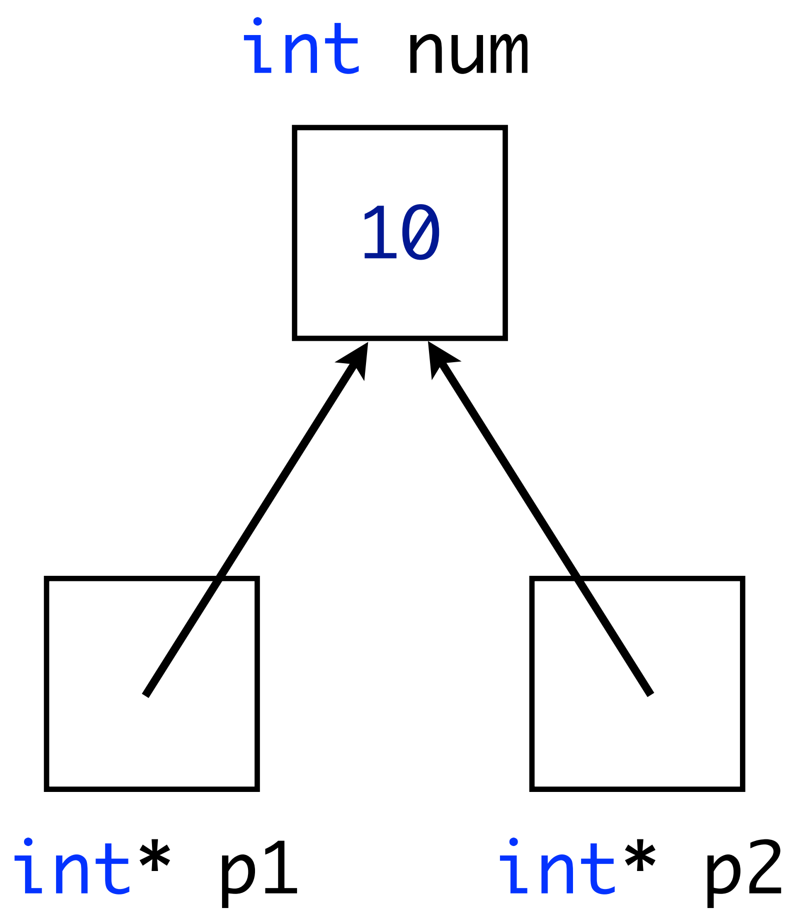
Pointer Example

Let's say we have these variables:

```
int num = 42;
int *p1, *p2;
```

Then we do some assignment:

```
p1 = &num;
p2 = p1;
*p2 = 10;
```



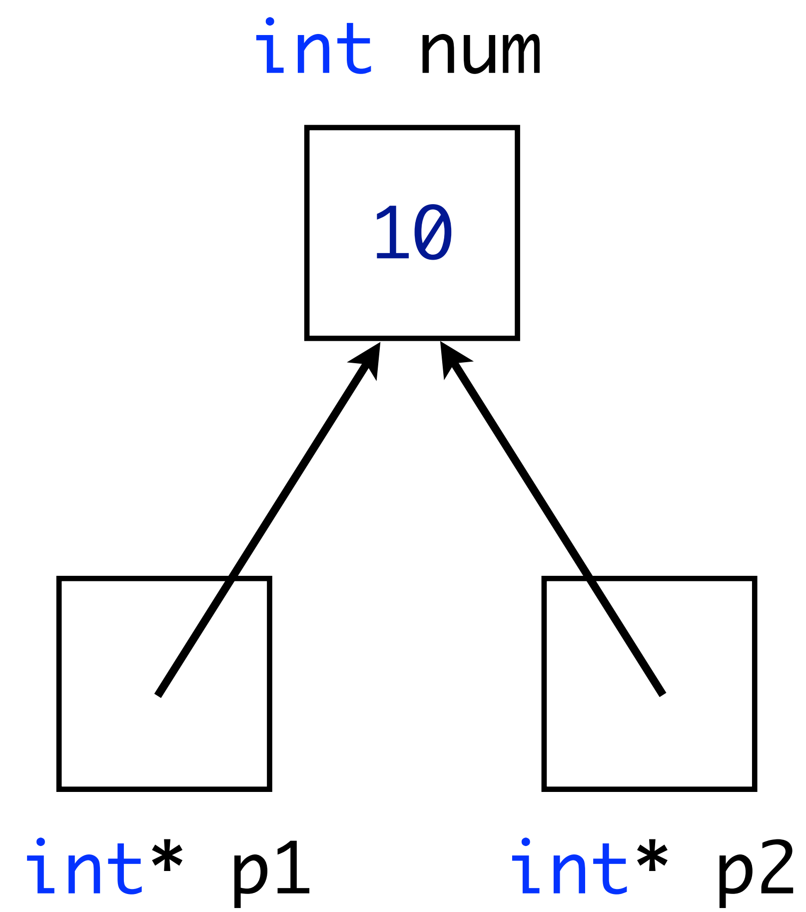
Pointer Example

Let's say we have these variables:

```
int num = 42;
int *p1, *p2;
```

Then we do some assignment:

```
p1 = &num;
p2 = p1;
*p2 = 10;
```



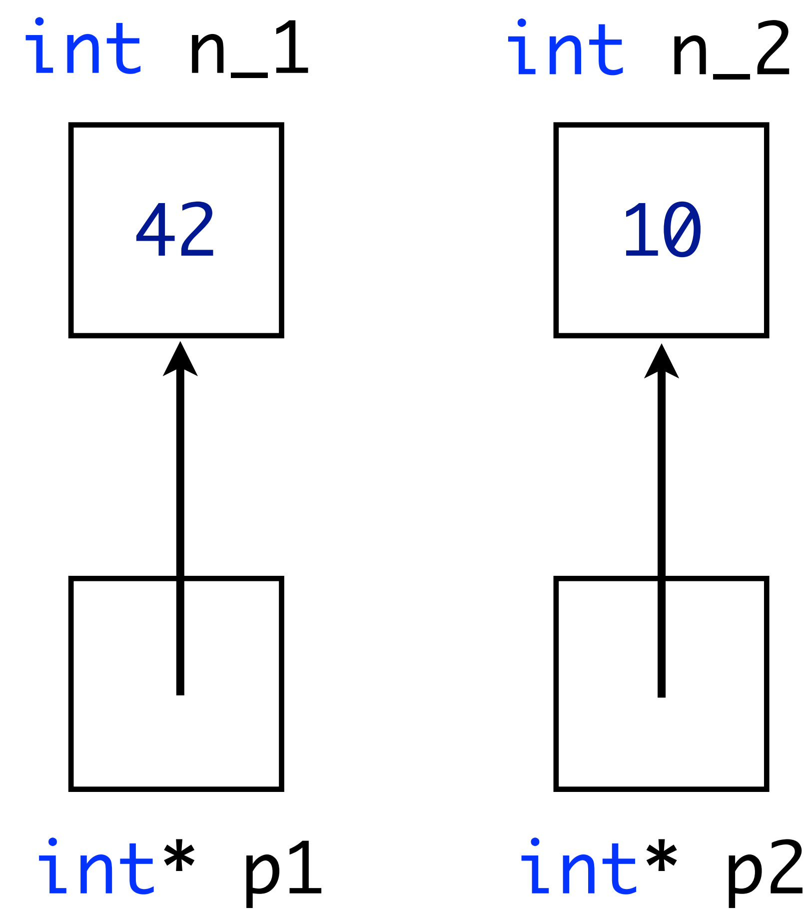
What does this statement output?

```
cout << num << " " << *p1 << " " << *p2 << endl;
// displays: 10 10 10
```

Another Example

Let's say we have these variables:

```
int n_1 = 42;
int n_2 = 10;
int* p1 = &n_1;
int* p2 = &n_2;
```



What happens when you do this:

```
p1 = p2;
```

Versus this?

```
*p1 = *p2;
```

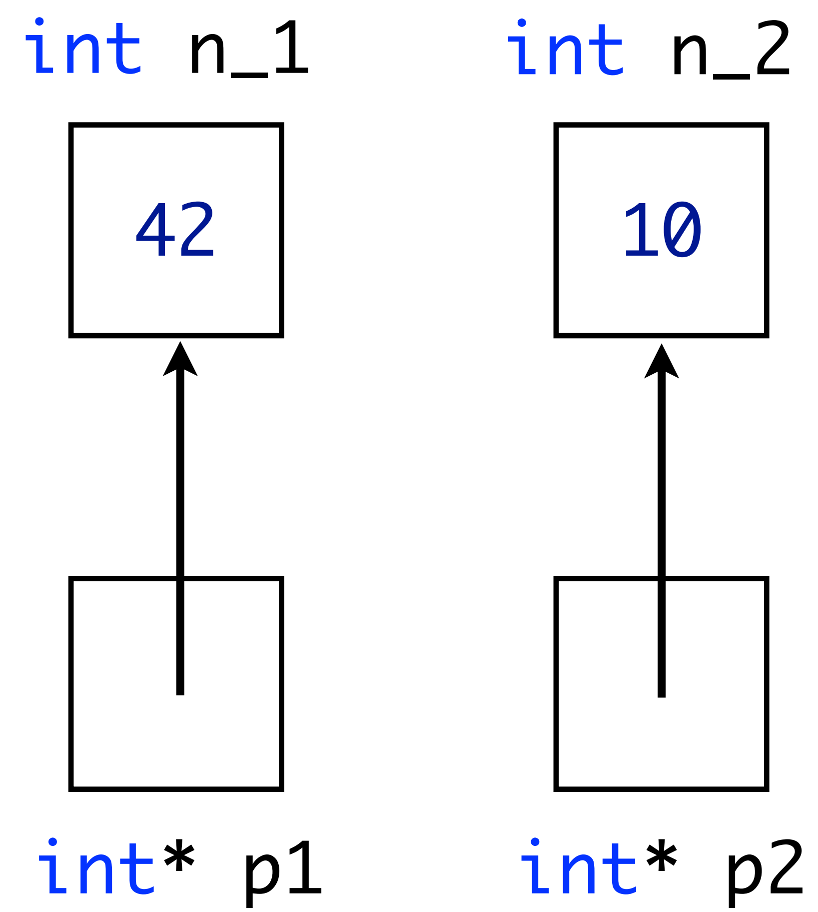
Another Example

Let's say we have these variables:

```
int n_1 = 42;
int n_2 = 10;
int* p1 = &n_1;
int* p2 = &n_2;
```

When we do this:

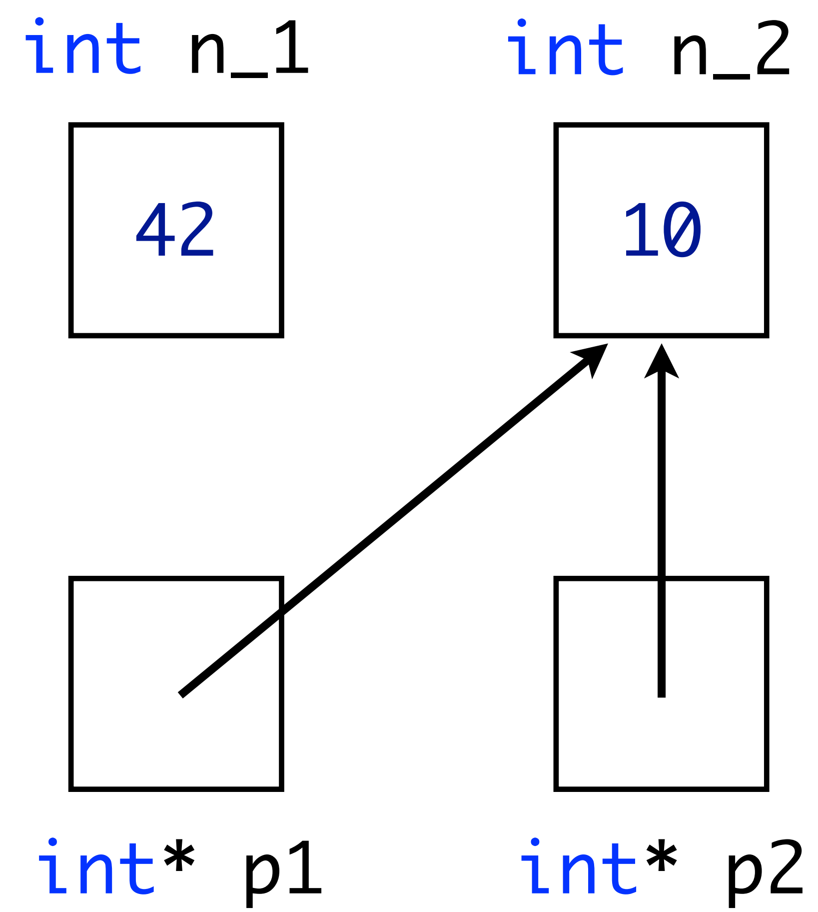
```
p1 = p2;
```



Another Example

Let's say we have these variables:

```
int n_1 = 42;
int n_2 = 10;
int* p1 = &n_1;
int* p2 = &n_2;
```



When we do this:

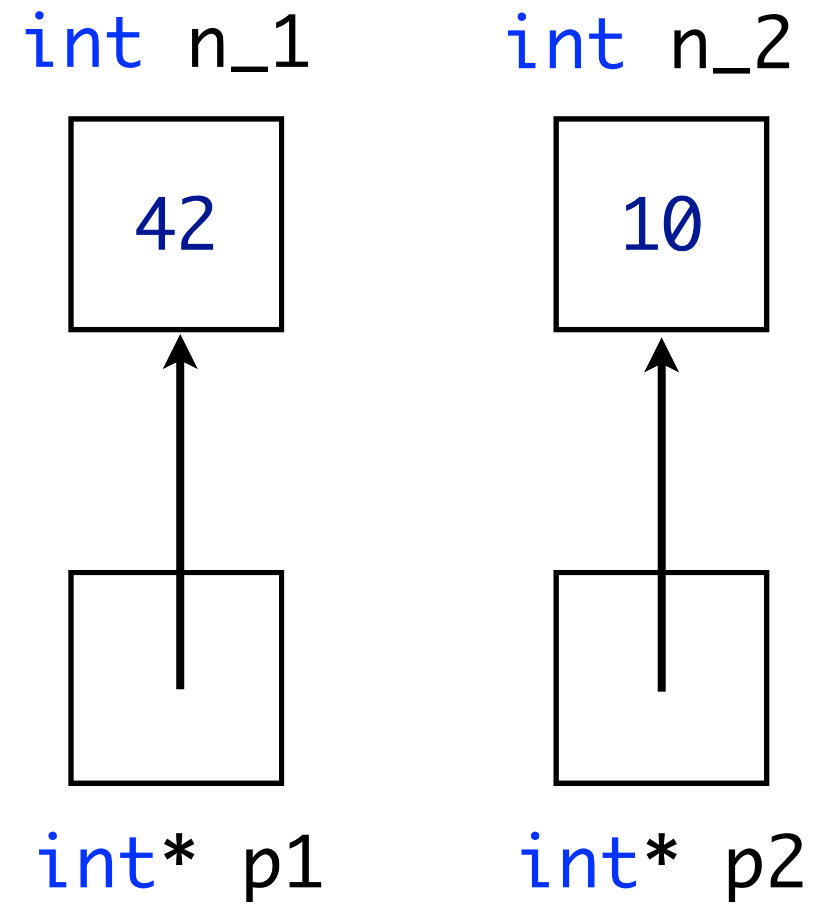
```
p1 = p2;
```

p1 now points at the same variable as p2...

Another Example

Let's say we have these variables:

```
int n_1 = 42;
int n_2 = 10;
int* p1 = &n_1;
int* p2 = &n_2;
```



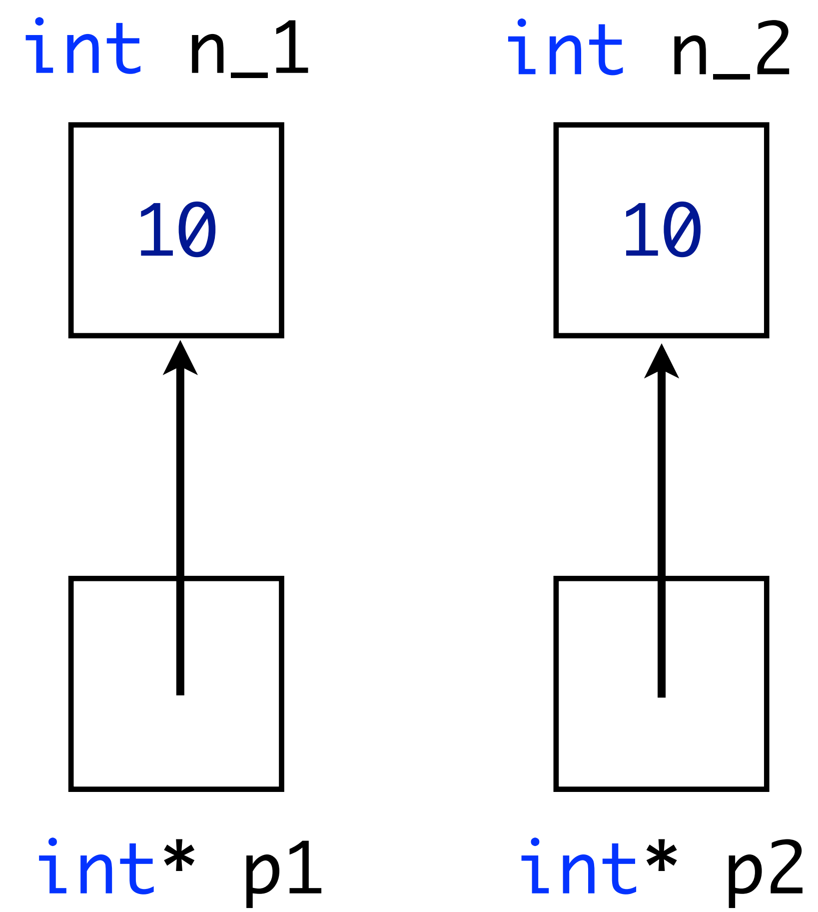
When we do this:

```
*p1 = *p2;
```

Another Example

Let's say we have these variables:

```
int n_1 = 42;
int n_2 = 10;
int* p1 = &n_1;
int* p2 = &n_2;
```



When we do this:

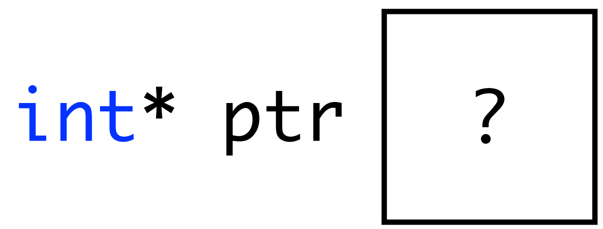
```
*p1 = *p2;
```

The value of the variable at which `p1` points changes to the value of the variable at which `p2` points...

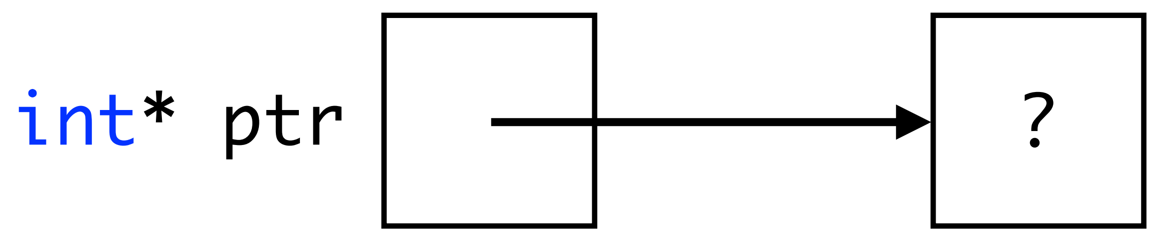
Dynamic Allocation

You can also use pointers with dynamically allocated values:

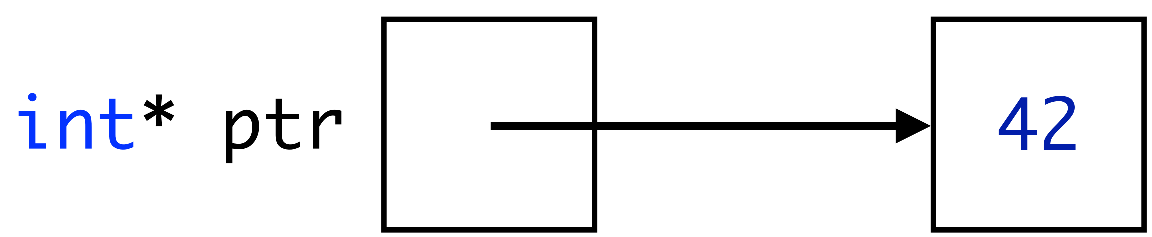
```
// an int pointer
int* ptr;
```




```
// point it at a new int
ptr = new int;
```




```
// give the int a value
*ptr = 42;
```



The new operator

Dynamic memory allocation is done using the new operator

- new allocates memory for a variable of the specified type
- it calls the appropriate constructor (when creating objects)
- if successful, the address of the newly created value is returned
- on failure, new throws a `bad_alloc` exception (ends your program unless handled)
- alternatively, you can use `new (nothrow)`, which simply returns `NULL` on failure

Examples:

```
// allocates a string using default constructor
string* str_ptr = new string;
// uses non-default constructor; returns NULL on failure
string* str_ptr = new (nothrow) string("Hi!");
```

The new operator

You can use new to request more than one variable at a time (an array):

```
// allocates an array of 100 ints
int* int_ptr = new int[100];
```

Simply specify how many values you want in brackets [ ]

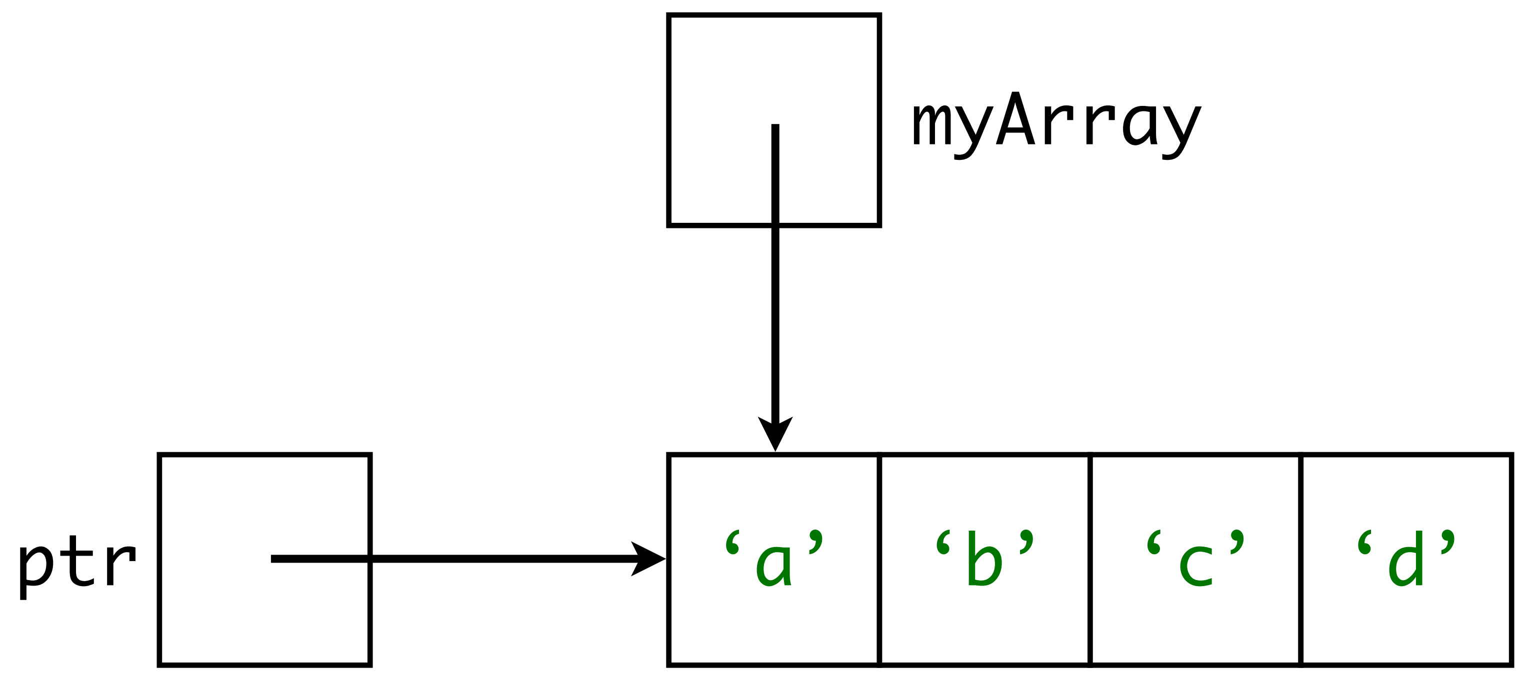
- this operator returns the address of the first element in the allocated block
- this version always uses the default constructor for objects

You can now use `int_ptr` just like you would a normal array...

Arrays are Pointers

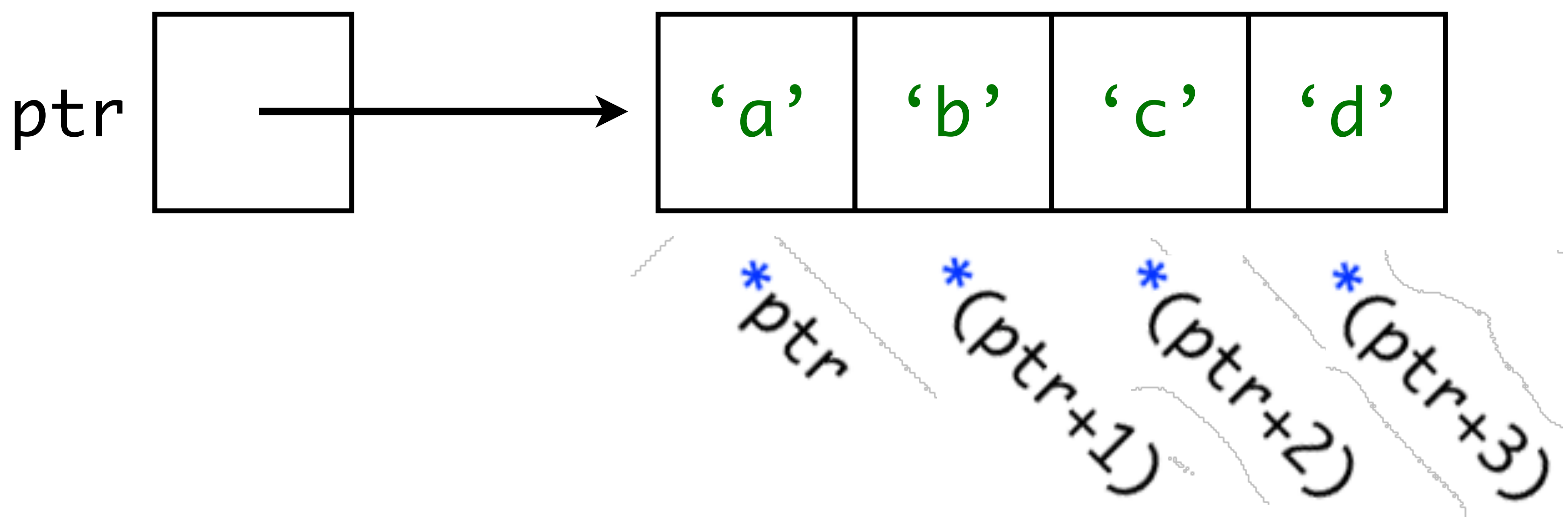
Array variables—statically or dynamically allocated—are just pointers

```
// myArray holds the address of the first element
char myArray[] = {‘a’, ‘b’, ‘c’, ‘d’};
char* ptr = myArray; // copy address into ptr
```

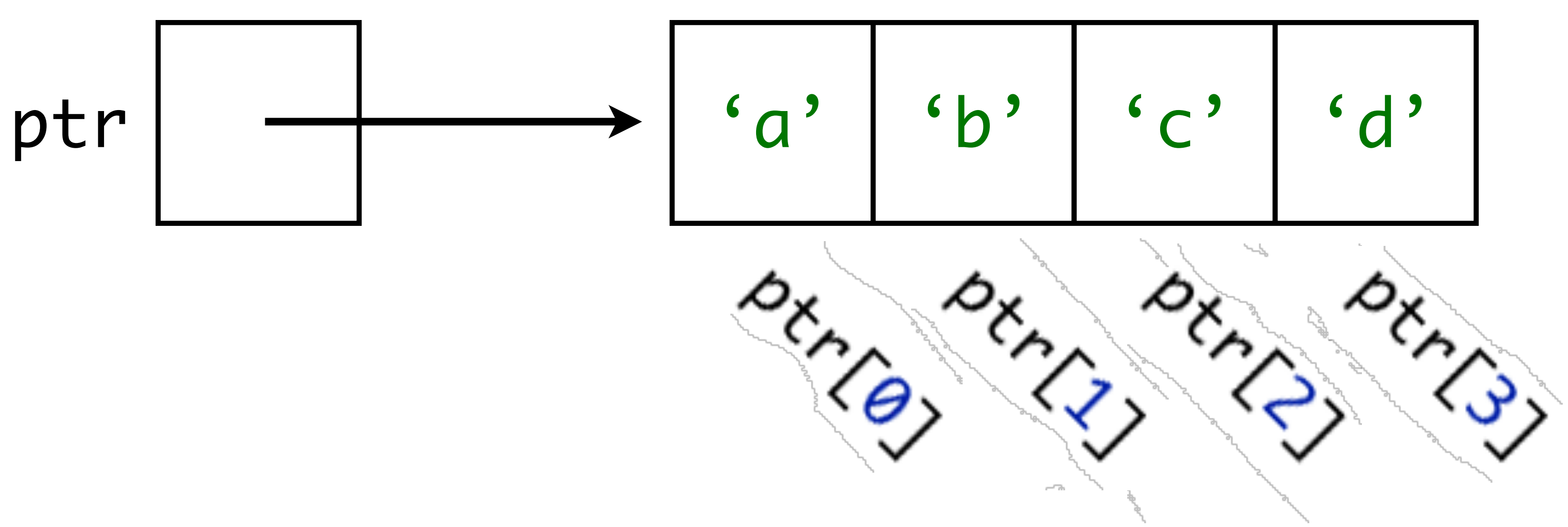


Array Indexes / Offsets

You can dereference each element by using an offset:



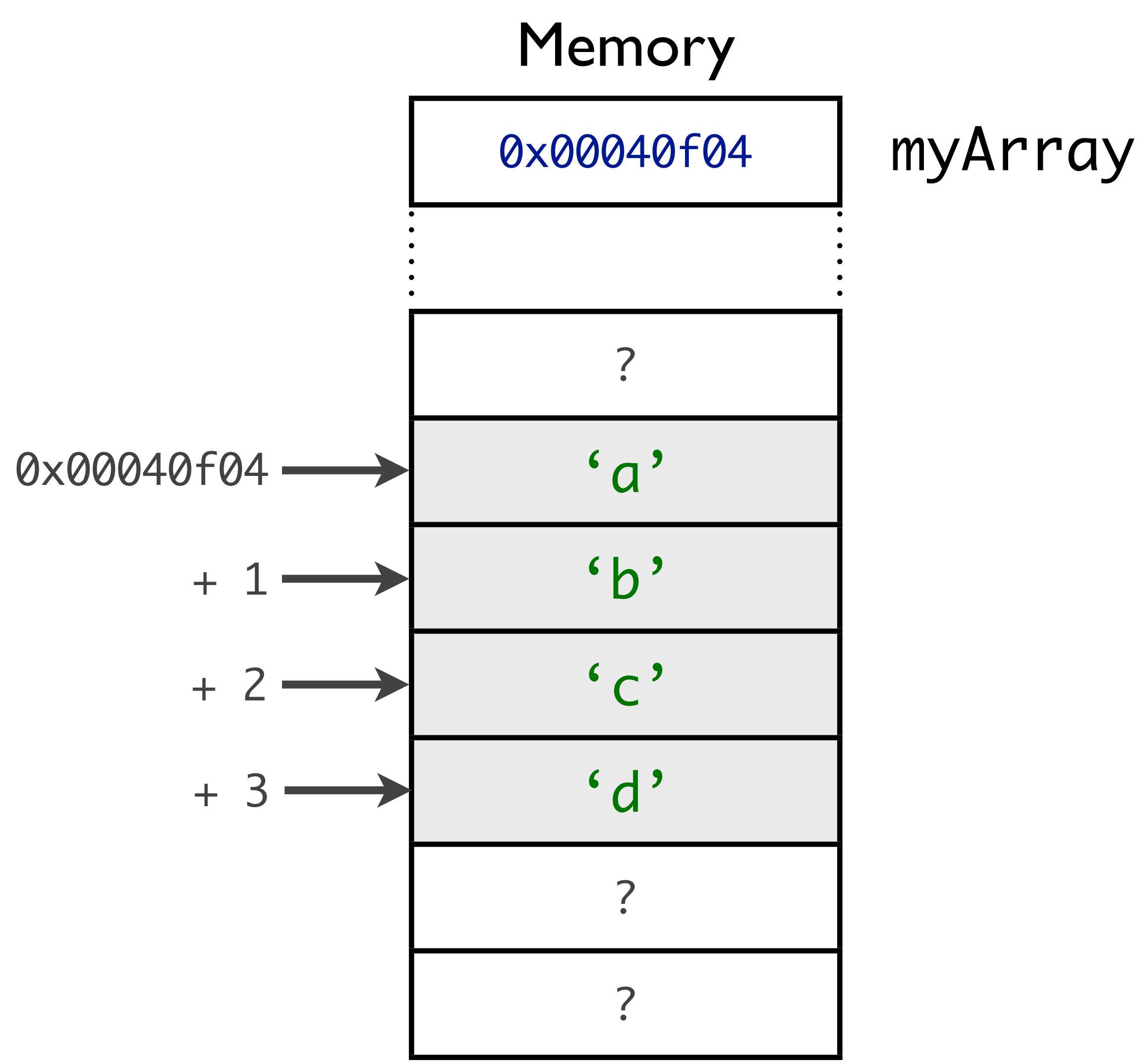
Luckily, there's a much easier way to dereference (you may recognize it):



Array Indexes / Offsets

Array variables—statically or dynamically allocated—are just pointers

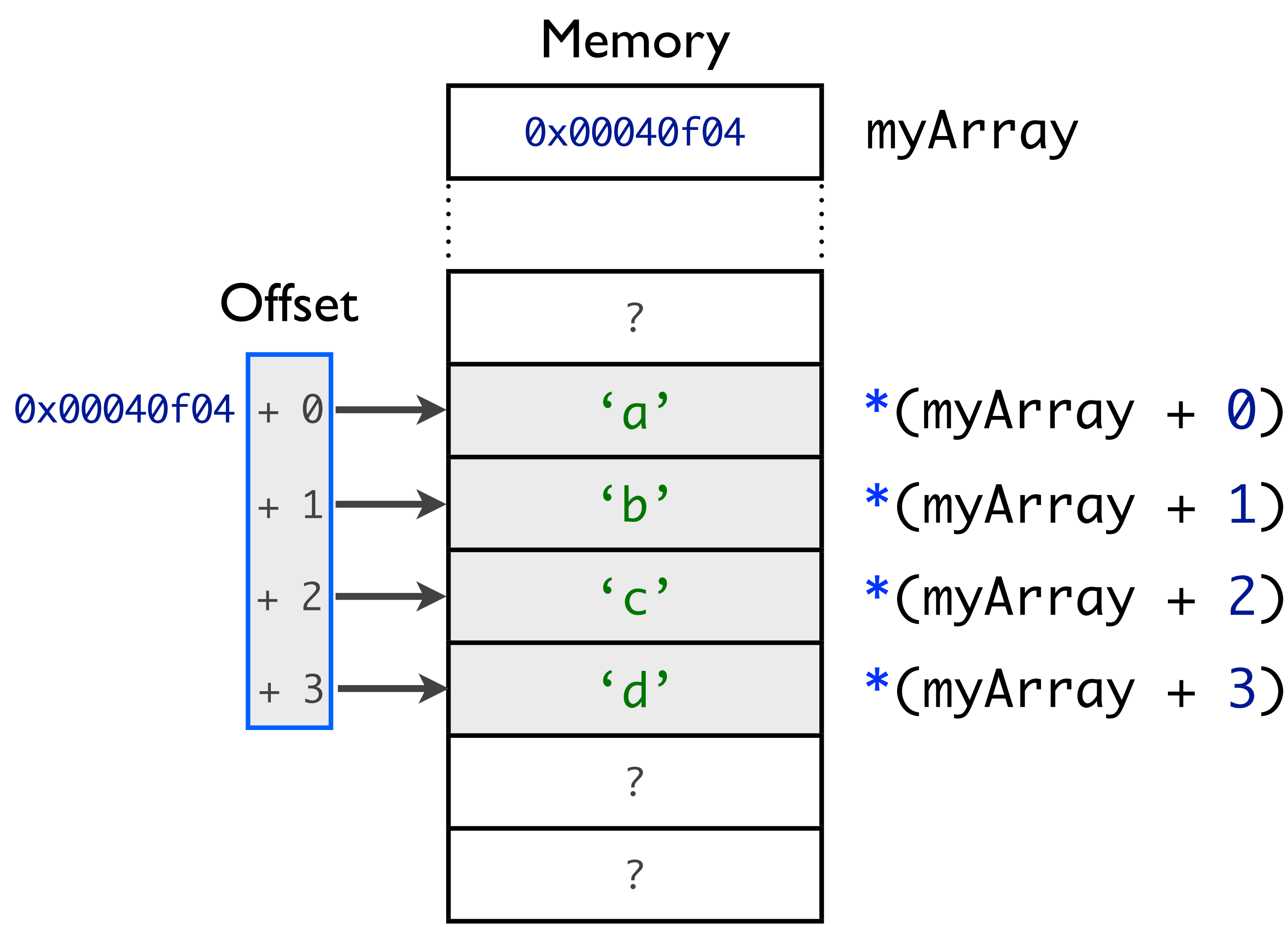
```
// myArray holds the address of the first element
char myArray[] = {'a', 'b', 'c', 'd'};
```



Array Indexes / Offsets

Array variables—statically or dynamically allocated—are just pointers

```
// myArray holds the address of the first element
char myArray[] = {'a', 'b', 'c', 'd'};
```

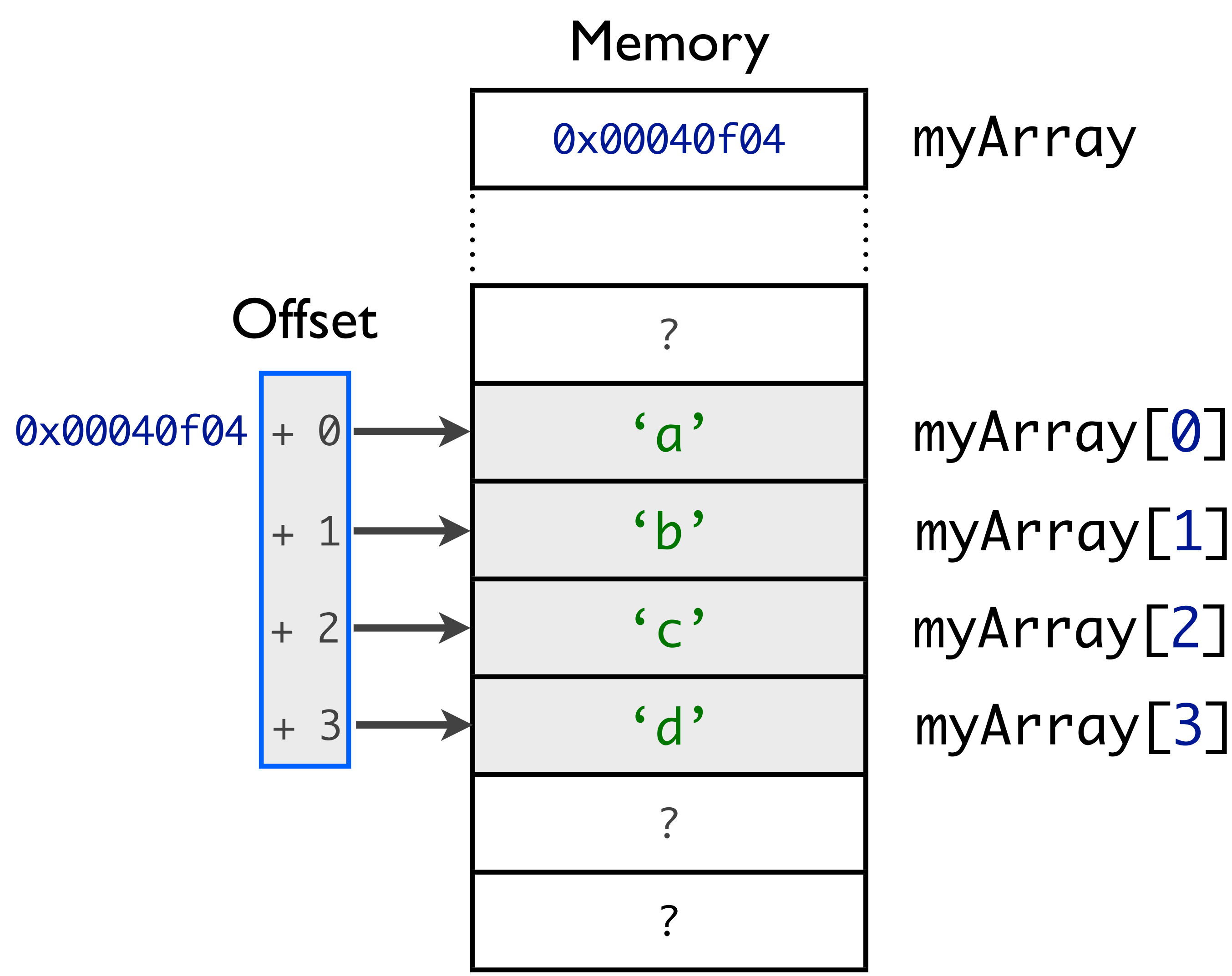


An element's index is the same as its offset!

Array Indexes / Offsets

Array variables—statically or dynamically allocated—are just pointers

```
// myArray holds the address of the first element
char myArray[] = {'a', 'b', 'c', 'd'};
```



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

Deallocating Dynamic Memory

Statically allocated variables are automatically cleaned up by C++

- such variables are *pushed* onto the stack when entering the scope in which they exist and are *popped* off the stack (their memory deallocated) when leaving

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

- dynamic memory is drawn off a memory area called the heap
- deallocating the memory makes it available for reuse later
- use the delete operator to deallocate a single value
- use the delete[] operator to deallocate an entire array

Forgetting to free the memory results in a memory leak

- it will also result in penalty to your grade =)

Deallocating Dynamic Memory

Example of deallocating a single dynamic-memory variable using delete:

```
// dynamically allocate an integer variable
int* int_ptr = new int;

// change the value to something... just for fun
*int_ptr = 42;

// deallocate the space used by the variable
delete int_ptr;
```

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 double[10];

// use the array just like you would any other...
for (int i = 0; i < 10; i++)
    dm_array[i] = i * 100;

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

Multidimensional Arrays

Multidimensional dynamically-allocated arrays require more work:

```
// dynamically allocate a 2D array of 10x10 ints
int** array = new int*[10];
for (int i = 0; i < 10; i++)
    array[i] = new int[10];

// deallocate the space used by the array
for (int i = 0; i < 10; i++)
    delete[] array[i];
delete[] array;
```

Notice the loops to allocate/deallocate the inner dimension!

Dangling Pointers

Once you `delete` a variable, you no longer have access to it

- the pointer will still point to the same spot in memory,
- but attempting to dereference it again is not allowed (crash)
- such a pointer is said to be left "dangling"
- setting pointers to `NULL` after you delete the memory is a good practice
- as is checking whether a pointer is `NULL` before dereferencing it

Be careful and think when using dynamic allocation!

- well, using care and thinking is probably a good idea when doing *anything*, really...