Get to Know Pointers

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- Pointer mistakes
- Pointers and arrays
- Pointer arithmetic
- Literal strings

Poking Around in Memory

- Pointers can let us read/write any memory location
- Very useful, but it also makes it easy to write some bad code.

```
int *p = (int *)12345678;
*p = 100;
int *q;
*q = 250;
```

Make up an address, then try to write there. Probably, you'd never do this on purpose..

But, if you forget to initialize a pointer, you may get the same effect.

- Kind of like buffer overflow errors with arrays.
  - Dereferencing a bad pointer can take you to who-knows-where in memory.

Pointers as Return Values

- Functions can return pointers.
  - This can be useful … we saw an example on pointer day 1.
- But, you have to consider …
  - will the destination address still be valid after you return?
- Here's a bad idea:

```
int *getLarger( int a, int b )
{
   if ( a > b )
     return &a;
   else
     return &b;
}
```

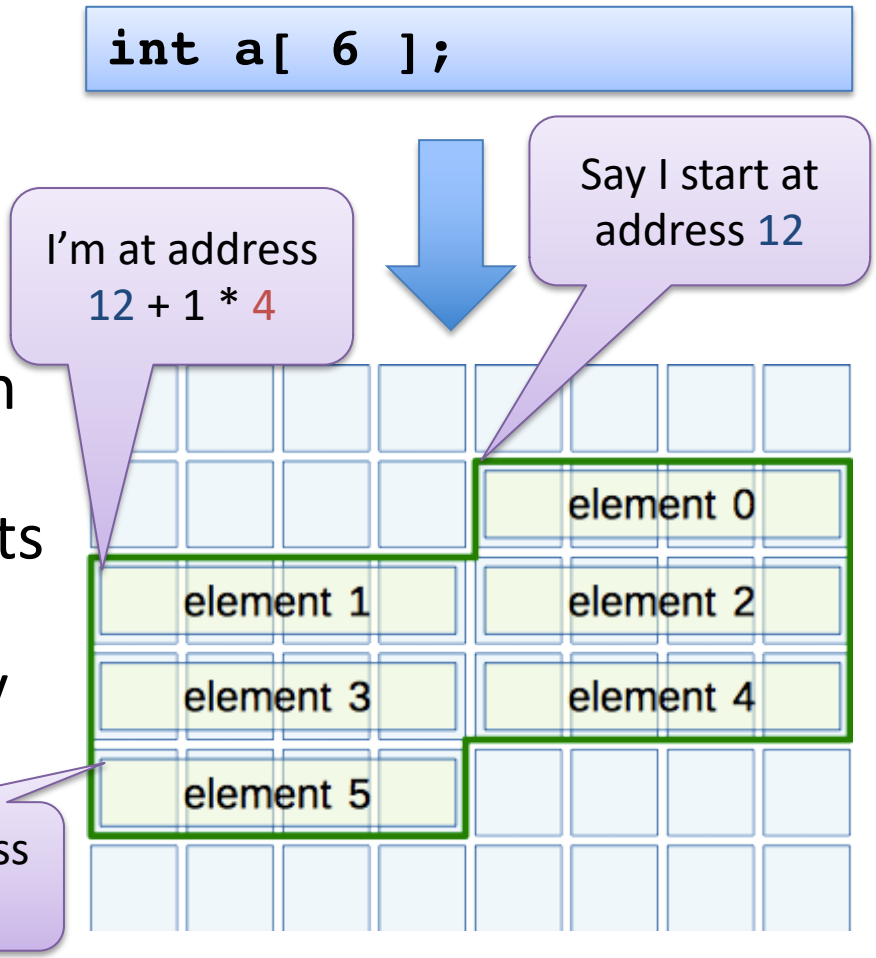
Here's your problem, these variables go away as soon as you return.

```
  int *p = getLarger( x, y );
  printf( "Sum: %d\n", *p );
```

This could be accessing stack space that's being used for something else now.

Array Internals

- Really, what is an array?
  - It's a sequence of values of the same type stored consecutively in memory.
- Why do these qualities matter?
  - If you know the size of each element
  - … and where the array starts in memory
  - … you can find the memory storing any element.



Array Internals

- So, to use an array, the compiler needs to keep up with:
  - How big each element is, so it can do indexing
    - That's implied by the element type
  - The location of the first element in memory
    - That sounds like a pointer
- And, the compiler will give you this pointer if you want it.

Arrays as Pointers

- We have a general rule.
  - If a is an array of some type T
    a evaluates to a pointer to the first element
    with a type of pointer to T (i.e., T *)

```
int a[ 6 ];
int *ap = a;

float b[ 100 ];
float *bp = b;

Whatever c[ 10 ];
Whatever *cp = c;
```

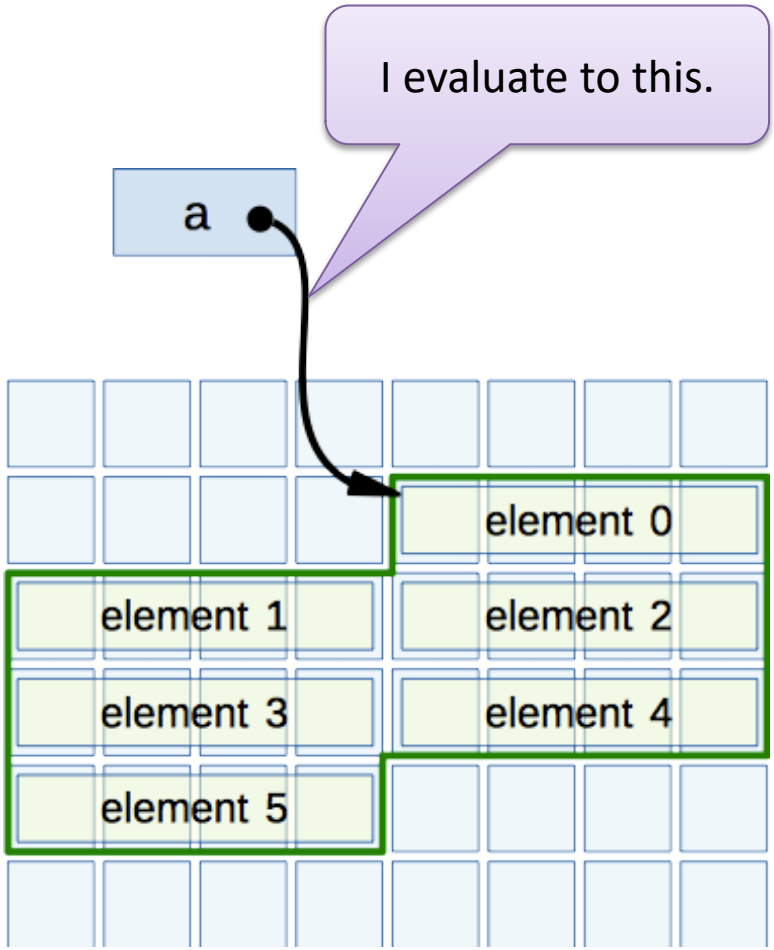
a evaluates to pointer to int.

b evaluates to pointer to float.

c evaluates to pointer to Whatever.

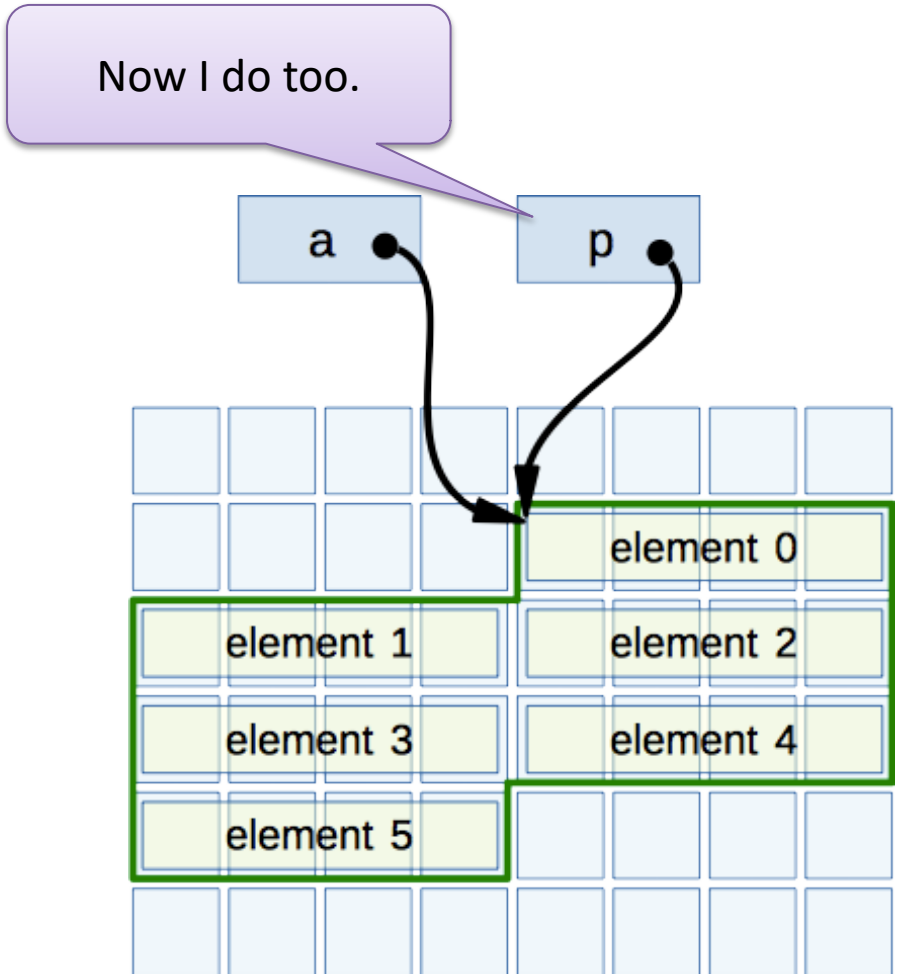
Arrays as Pointers … Illustrated!

```
int a[ 6 ];
```



Arrays as Pointers … Illustrated!

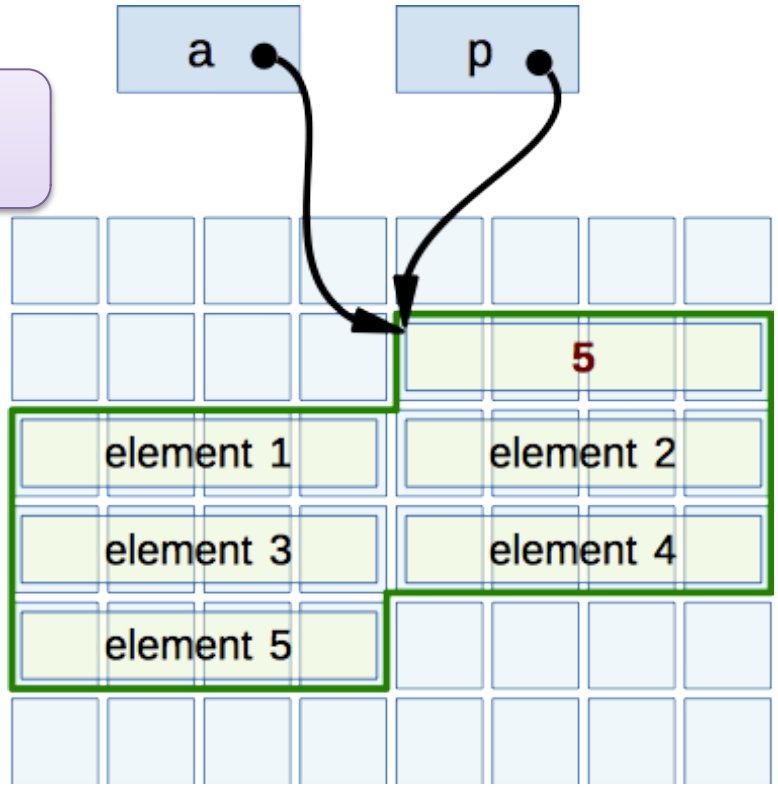
```
int a[ 6 ];

int *p = a;
```



Arrays as Pointers … Illustrated!

```
int a[ 6 ];

int *p = a;

a[ 0 ] = 5;
```

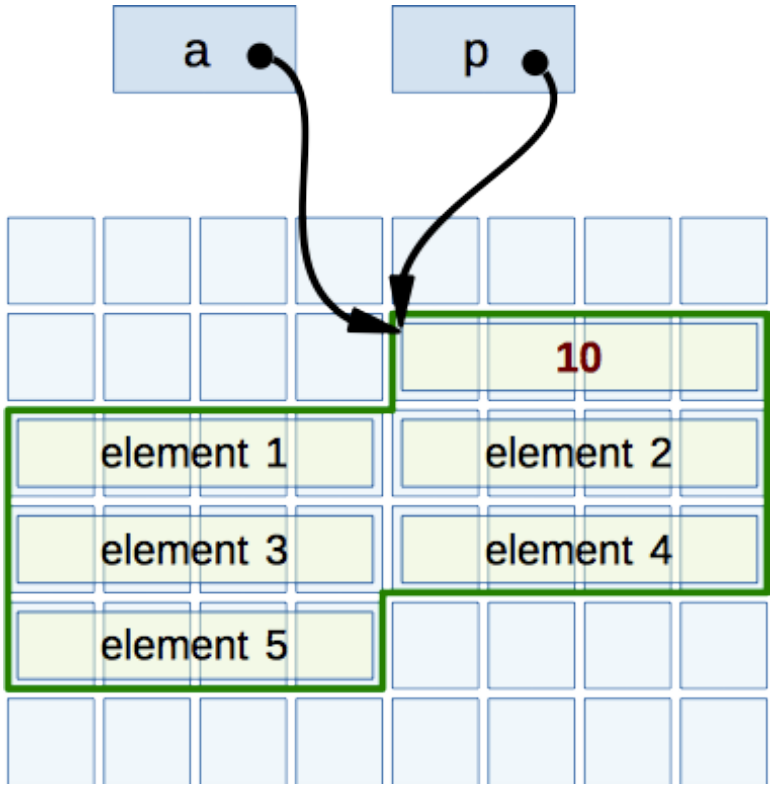
I can change the first element.



Arrays as Pointers … Illustrated!

```
int a[ 6 ];

int *p = a;

a[ 0 ] = 5;

*p = 10;
```

Me too.

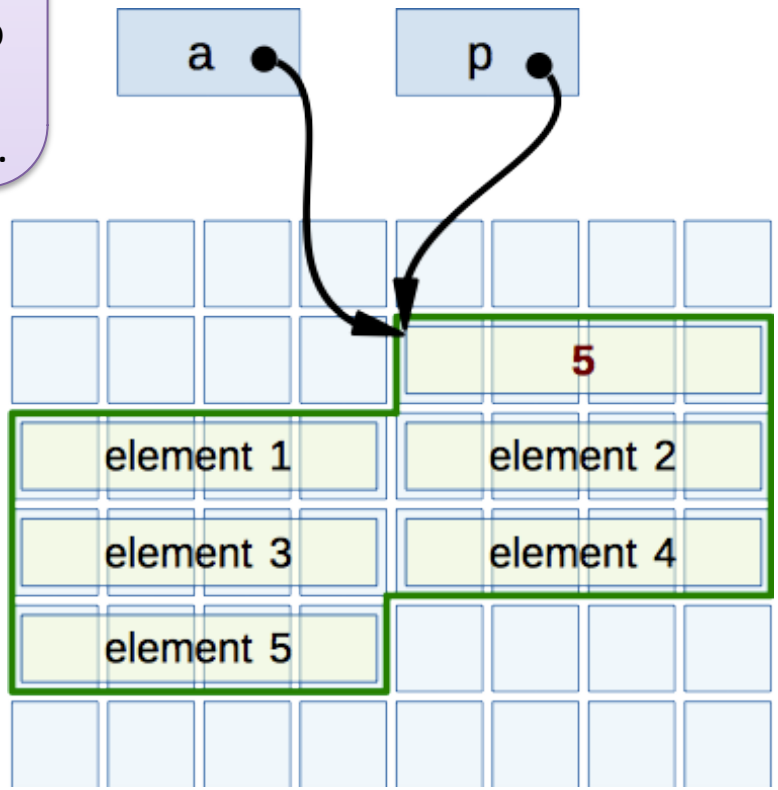


Arrays as Pointers … Illustrated!

```
int a[ 6 ];

int *p = a;

a[ 0 ] = 5;

*p = 10;

*a = 5;
```

Legal?

Sure. The array name evaluates to a pointer, so we can dereference it.



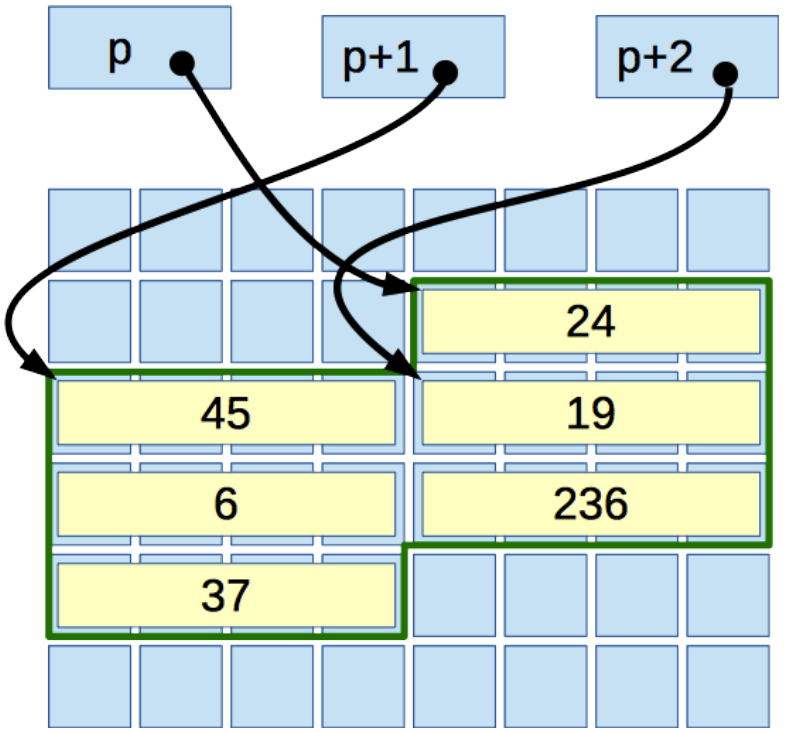
Arrays as Pointers

- Useful? Well no, not very useful … yet.
- This just gives us two ways of getting to the first array element.

- But, what if we could move this pointer around in memory?
  - Maybe we could get to other elements of the array.

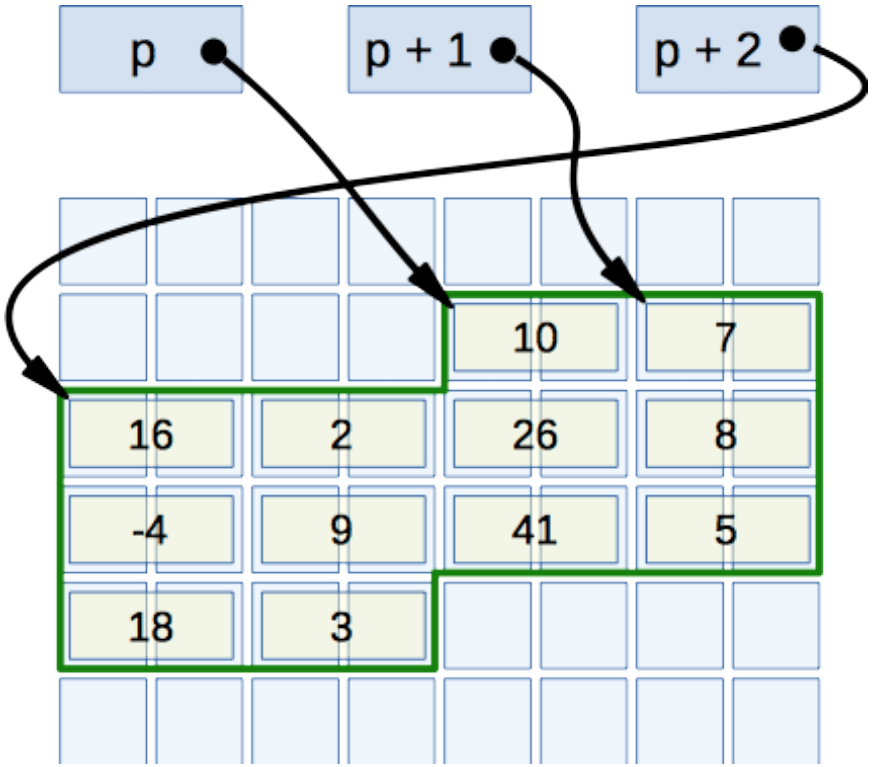
Pointer Arithmetic

- We can do (some) math on pointers, and it behaves in a very useful way.
  - It moves by multiples of sizeof for the type of object pointed to.
  - Adding 1 to an int pointer moves ahead 4 bytes.
  - Adding 2 to an int pointer moves ahead 8 bytes.
  - Adding 3 to an int pointer … well, you get the idea.



Pointer Arithmetic

- This works for any type
  - Adding to a short pointer moves by 2 bytes at a time.
  - Adding to a long pointer moves by 8 bytes at a time.
  - Adding to a char pointer moves one byte at a time.
  - …



Pointer Arithmetic

- We can see the effects of pointer arithmetic if we print the pointer's value.

```
char ca[ 10 ];
int ia[ 10 ];
double da[ 10 ];

char *cp = ca;
int *ip = ia;
double *dp = da;

printf( "%p %p %p\n", cp,     ip,     dp     );
printf( "%p %p %p\n", cp + 1, ip + 1, dp + 1 );
```

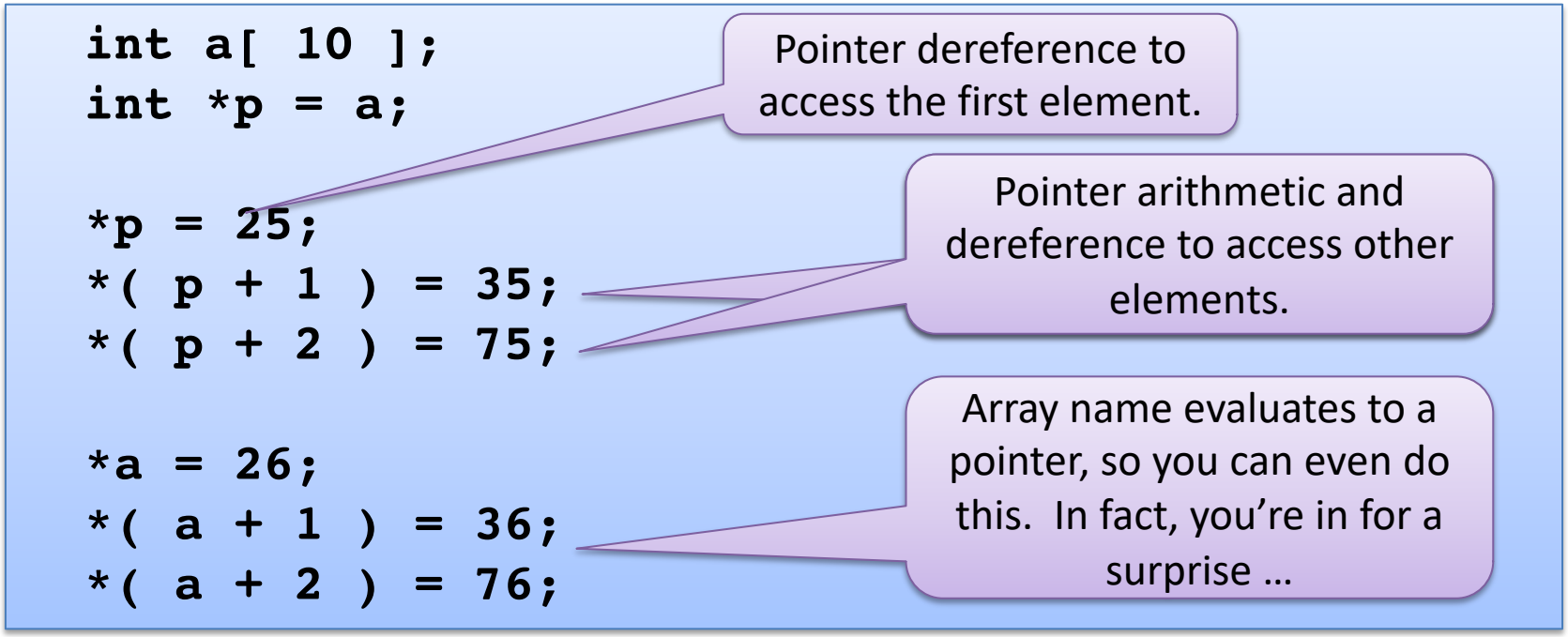
I'm 1 byte later.

I'm 4 bytes later.

I'm 8 bytes later.

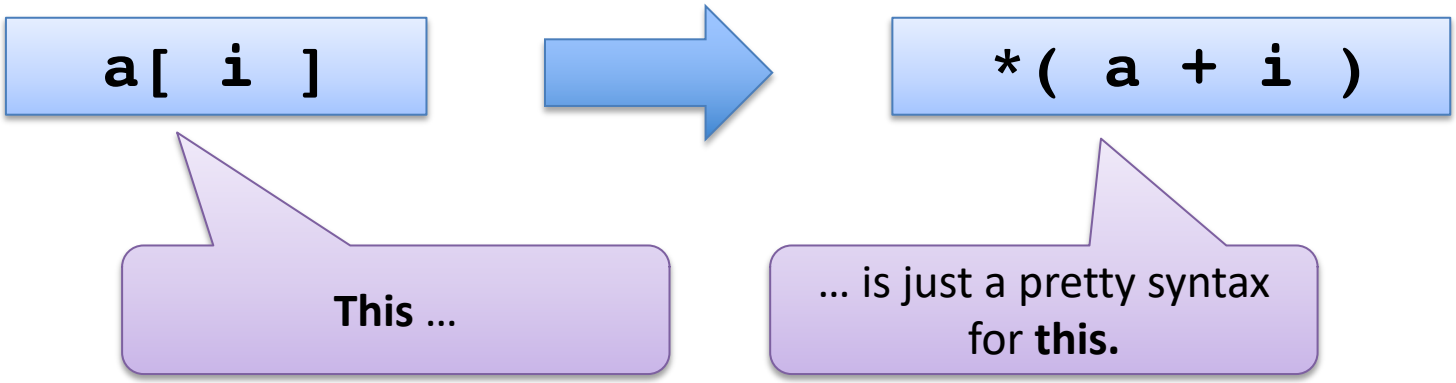
Array Elements via Pointers

- This behavior makes it easy to access nearby elements of the same type.

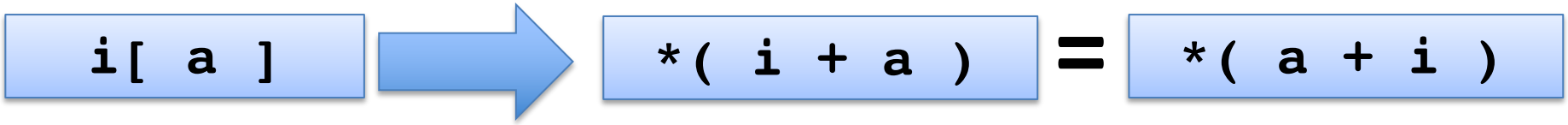


Array Indexing Explained

- This is what array indexing means in C

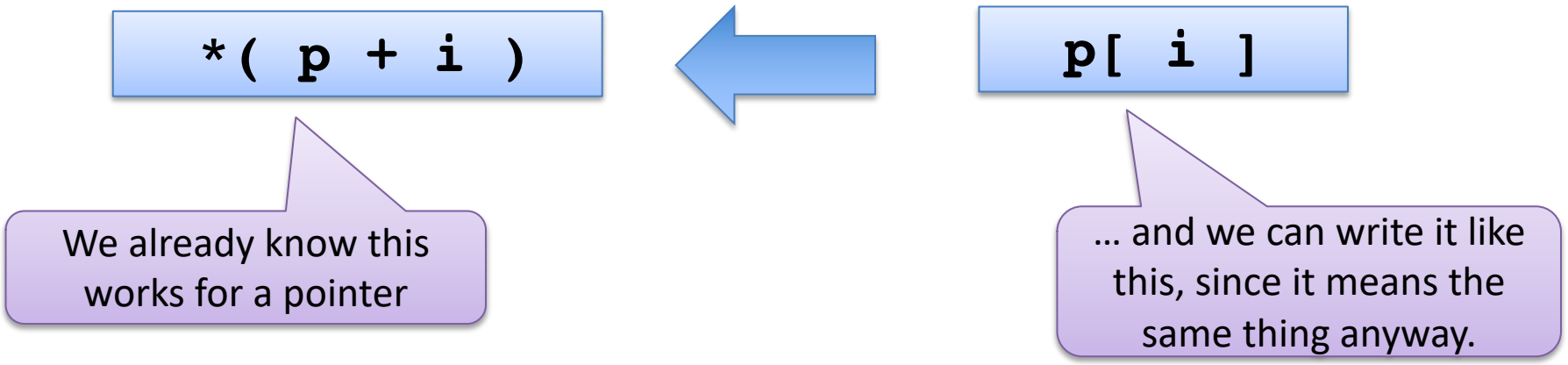


- Addition is commutative, so this is why we can write indexing backward:



Indexing With a Pointer

- If `a[i]` is just a nice way of writing `*(a+i)` …
- … and `*(p+i)` works for any pointer `p` … then:



- Arrays and pointers **aren't** the same thing,
  - But, in lots of situations, we can use them interchangeably.

Passing Arrays as Pointers

- So, this gives us an alternative way to pass arrays to functions (or you might think so)
  - Just pass the address of the first element.

```
void g( int *p )
{
  p[ 0 ] += 1;
  p[ 1 ] += 2;
}
```

I'm a pointer to the first element.

You can index from me, like I was an array.

```
  int myList[ ] = { 1, 4, 9 };
  g( myList );
```

I evaluate to the address of the first element.

Array Parameters Explained

- But, this is secretly how we've been passing arrays all along.
  - We've just been using an alternative syntax.

```
void g( int p[] )
{
  p[ 0 ] += 1;
  p[ 1 ] += 2;
}
```

When you ask for this …

```
void g( int *p )
{
  p[ 0 ] += 1;
  p[ 1 ] += 2;
}
```

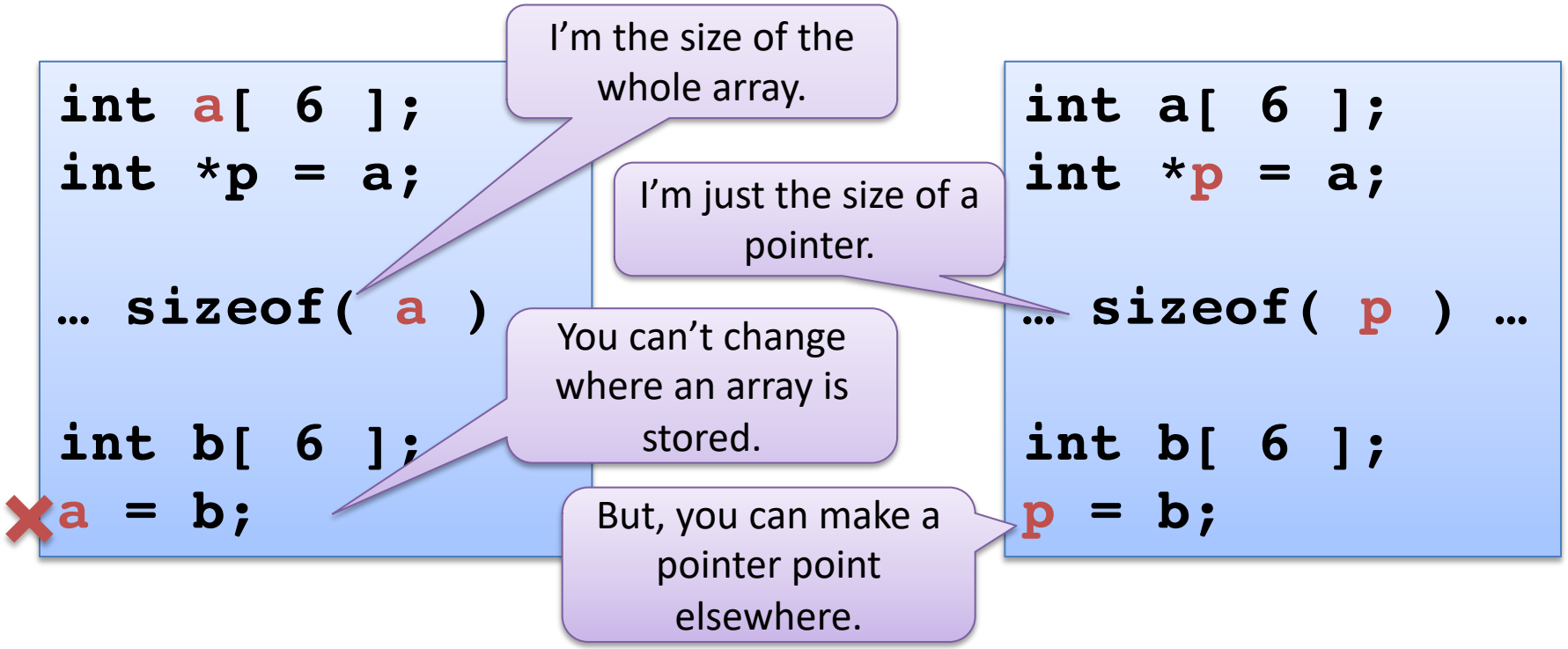
… the compiler does this.

```
  int myList[ ] =
      { 1, 4, 9 };
  g( myList );
```

```
  int myList[ ] =
      { 1, 4, 9 };
  g( myList );
```

Arrays Aren't Pointers

- You might start to think arrays and pointers are the same thing … but you'd be wrong.
- We can see a few situations where they don't behave exactly the same:



Passing an Array Offset

- Array Parameters are passed by pointer
- Any pointer will do, it doesn't have to be at the start of an array.

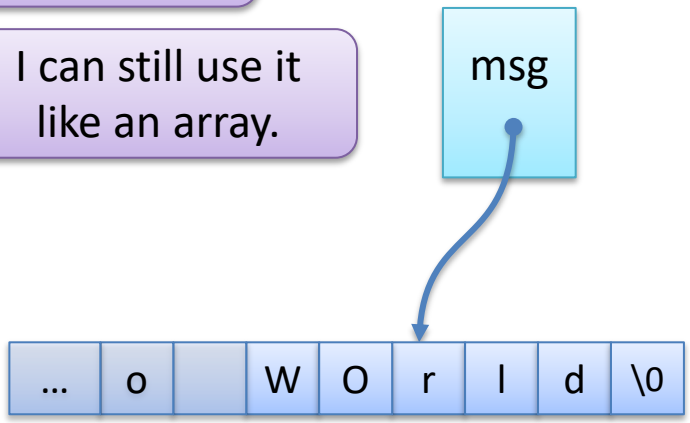
```
void capitalize( char *msg )
{
  for ( int i = 0; msg[i]; i++ ) {
    if ( msg[i] >= 'a' && msg[i] <= 'z' )
      msg[i] = msg[i] – 'a' + 'A';
  }
}
```

I'll take any char pointer.

I can still use it like an array.

```
  char str[] = "hello world";
  capitalize( str + 6 );
```

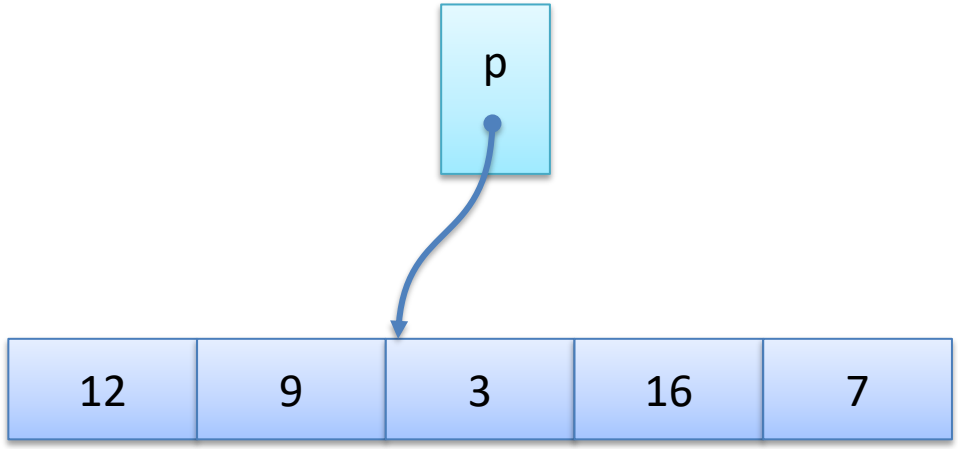
Here, just work on the last part of this array.



Pointer Arithmetic

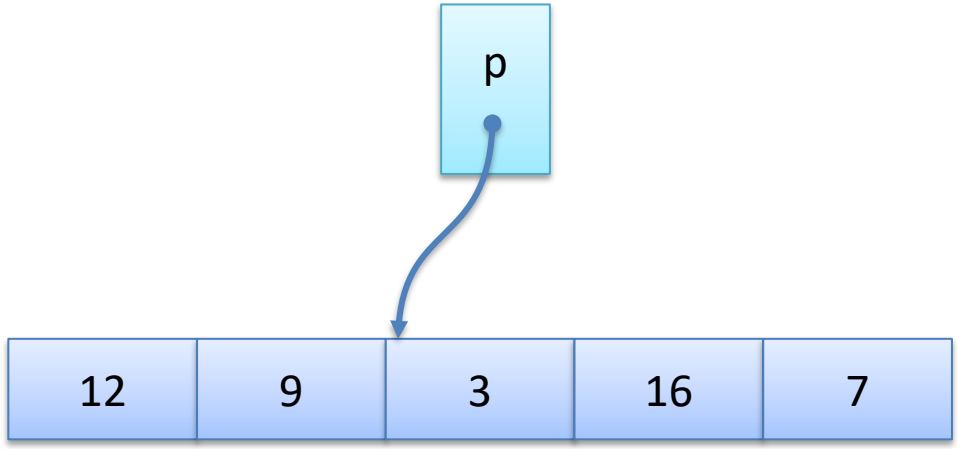
- We can use ++ and -- operators with pointers, to move forward or backward by one element.
  - Often combined with dereferencing.
  - Consider, what will each of these do?

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



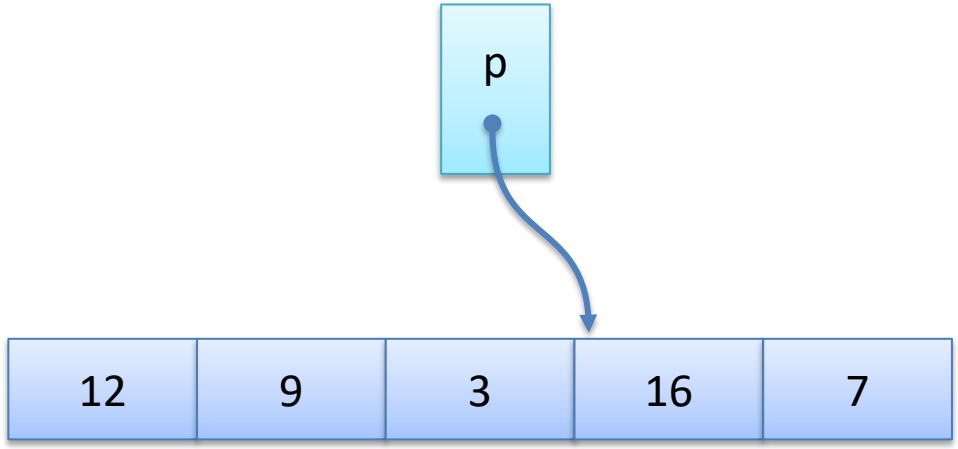
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



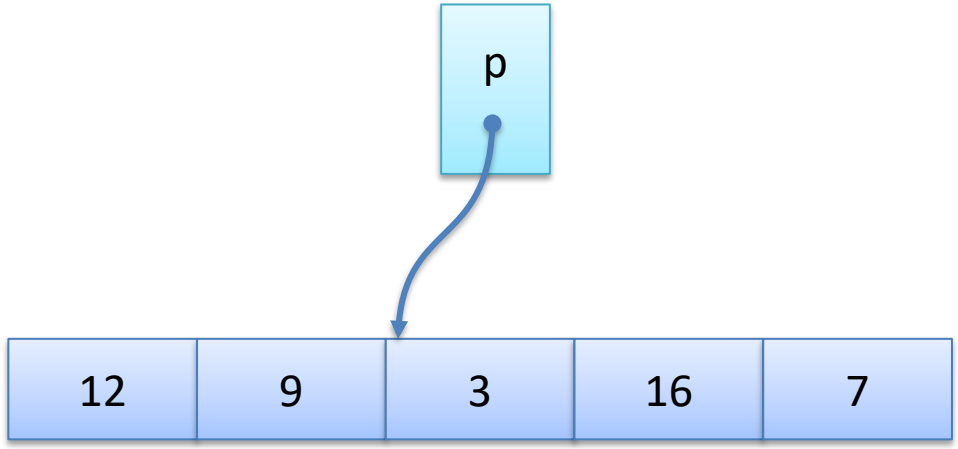
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



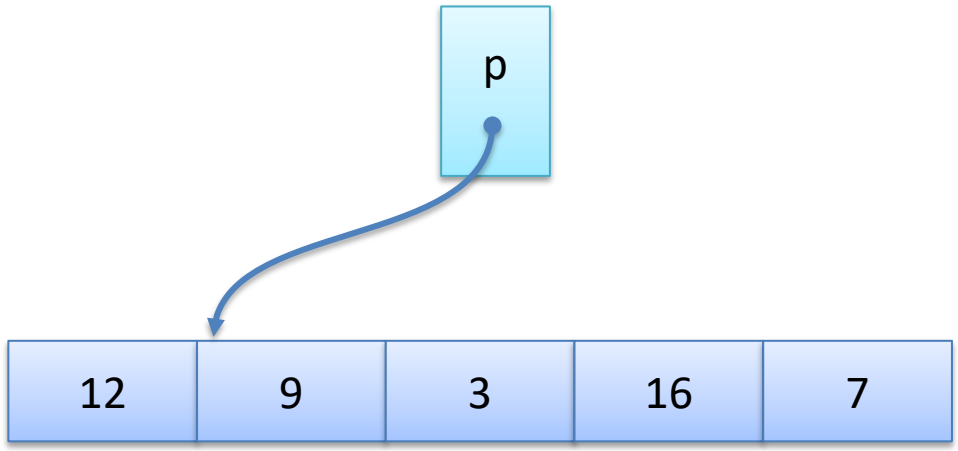
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



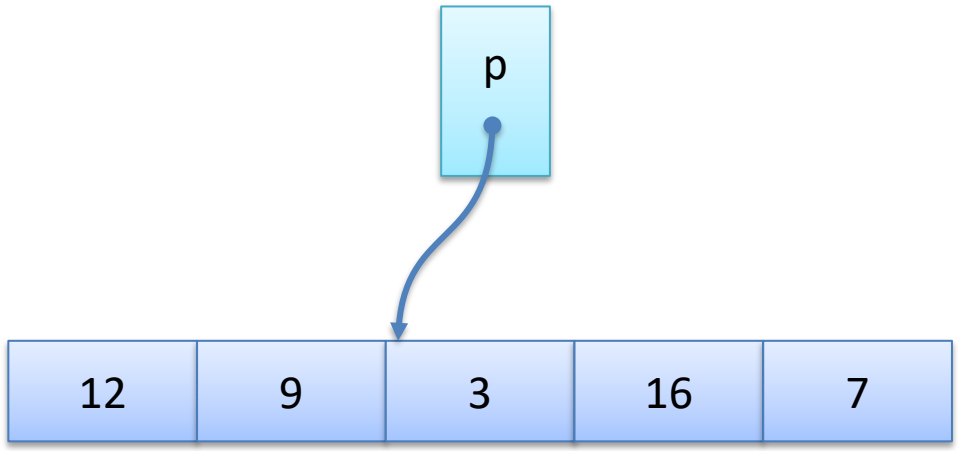
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



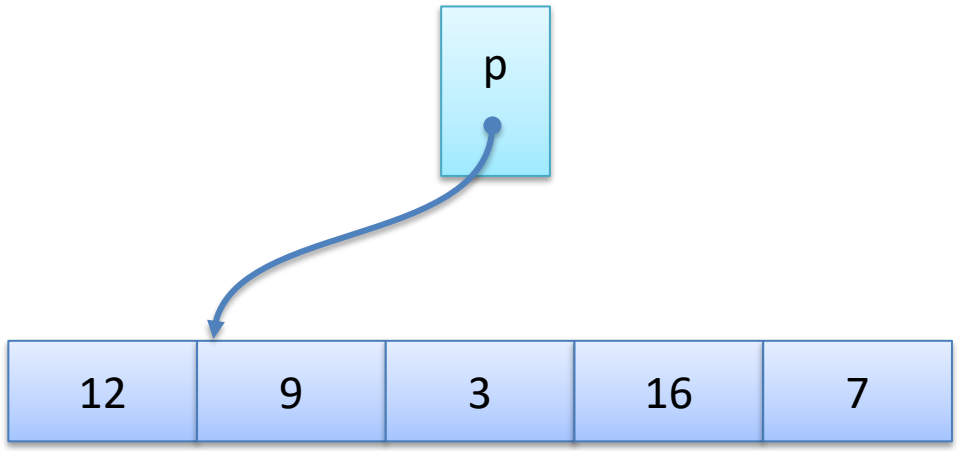
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



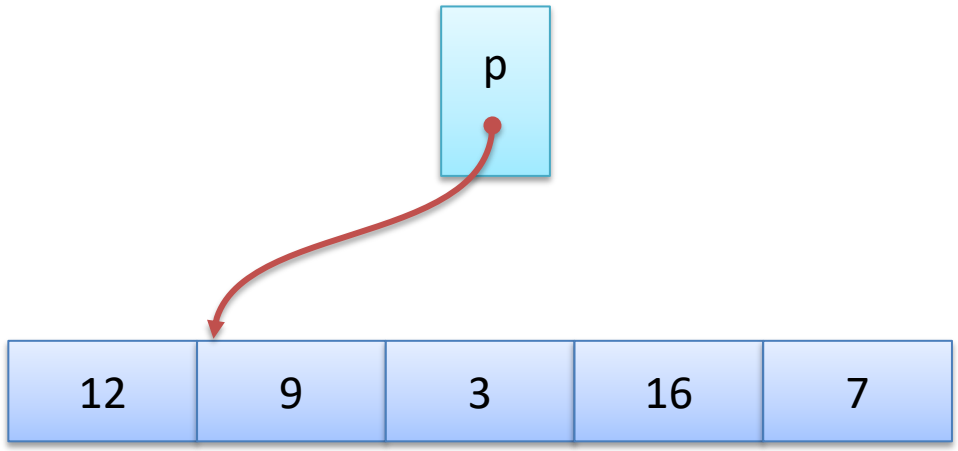
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



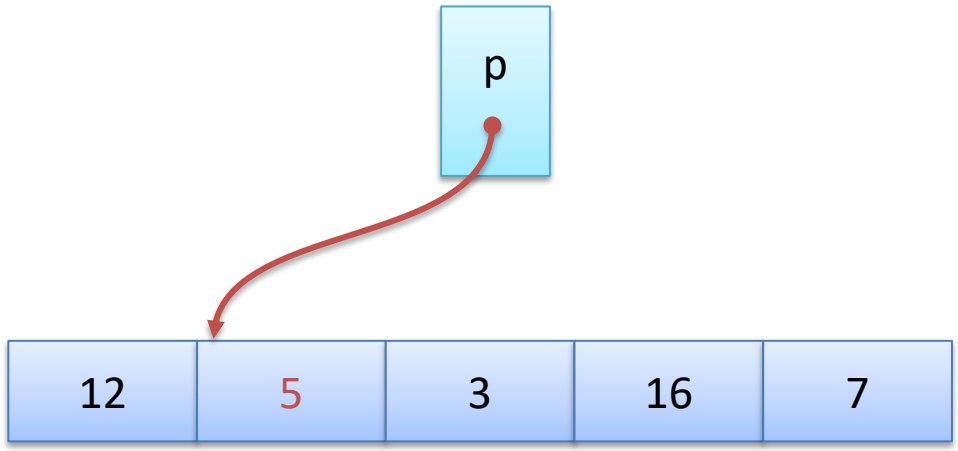
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



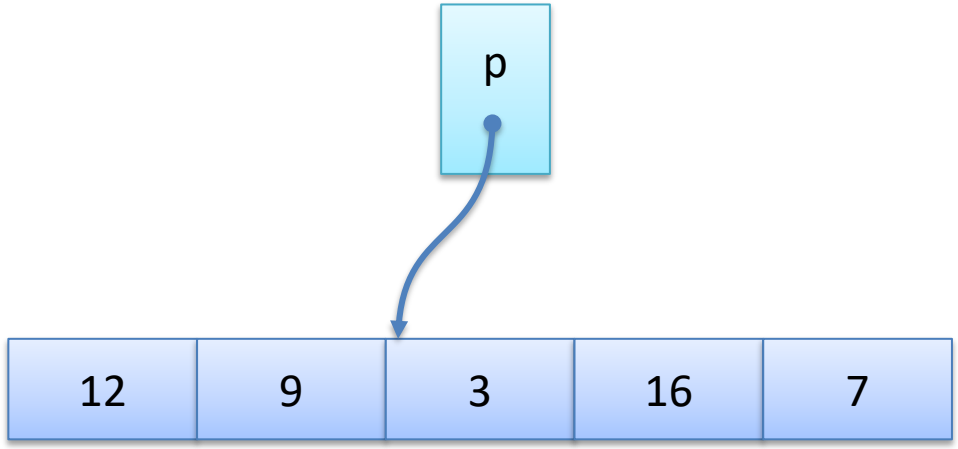
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



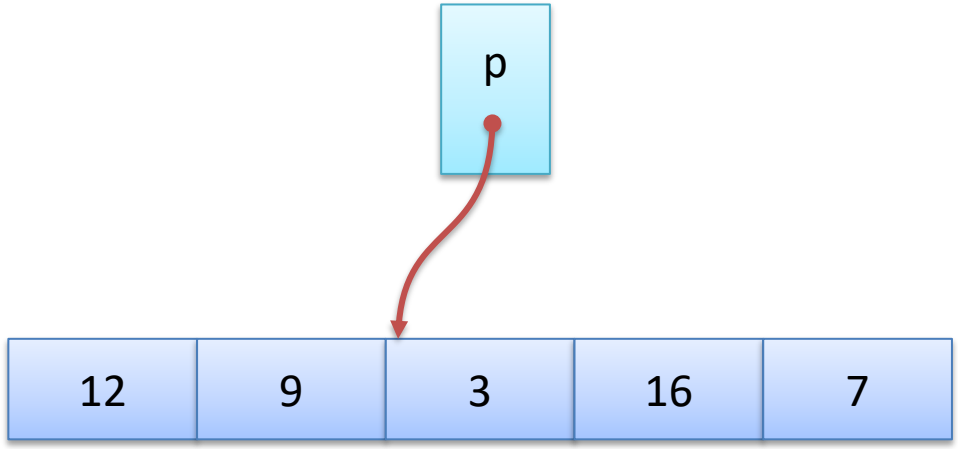
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



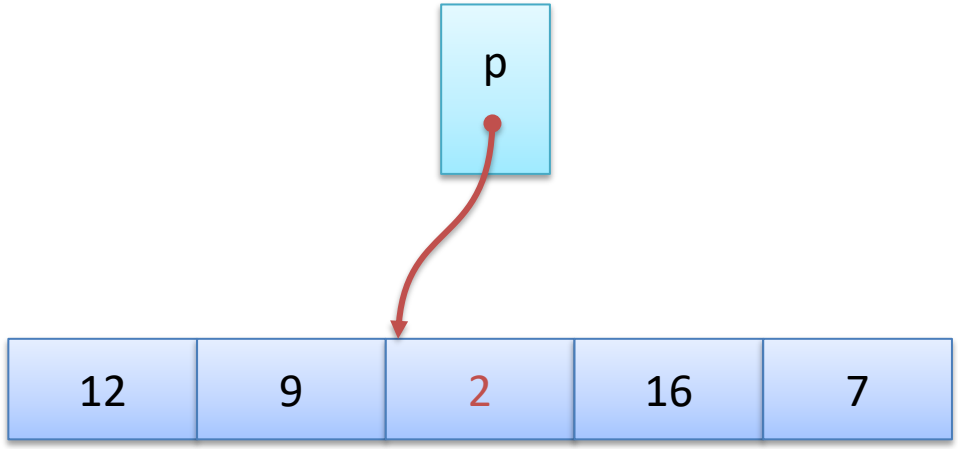
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



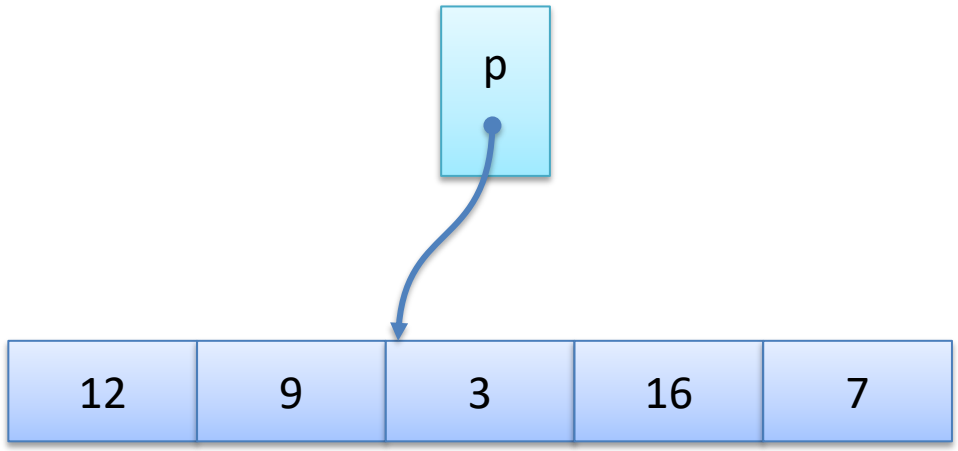
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



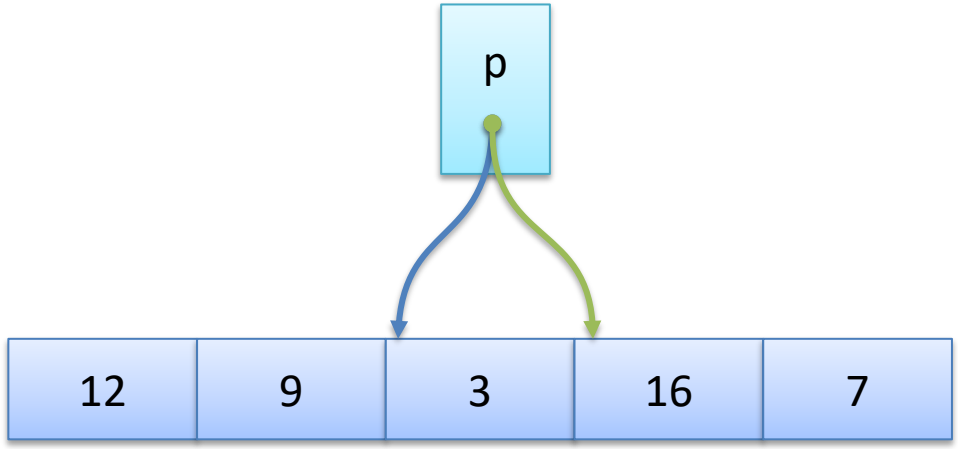
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



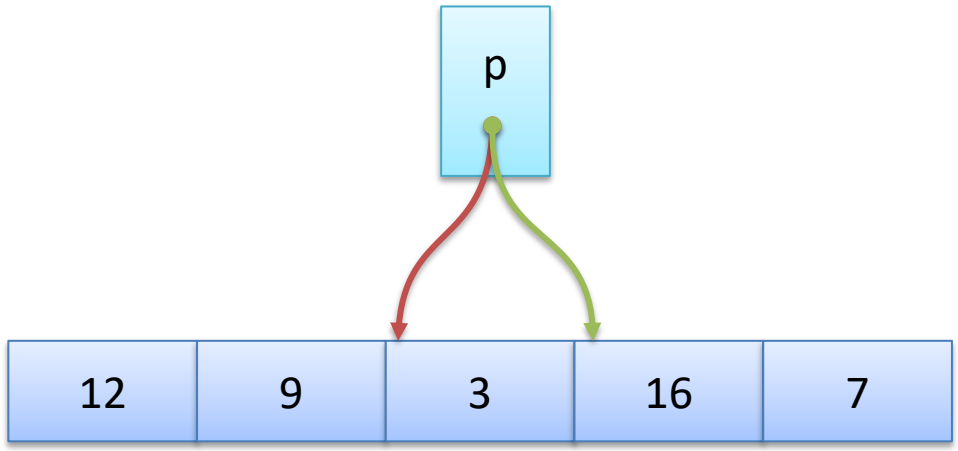
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```

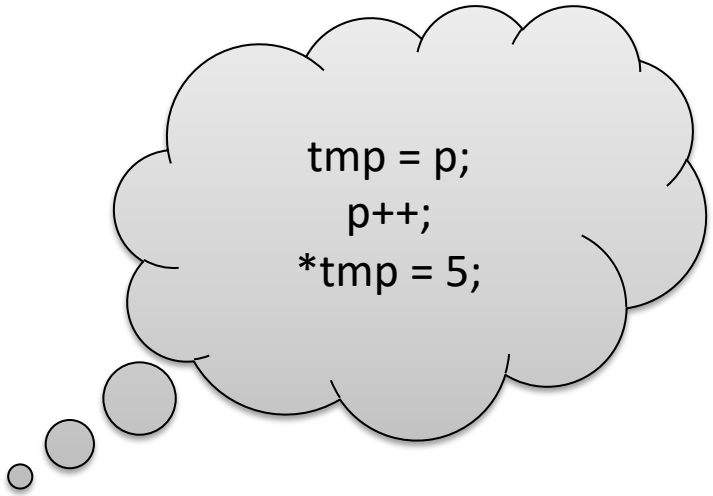


Pointer Arithmetic

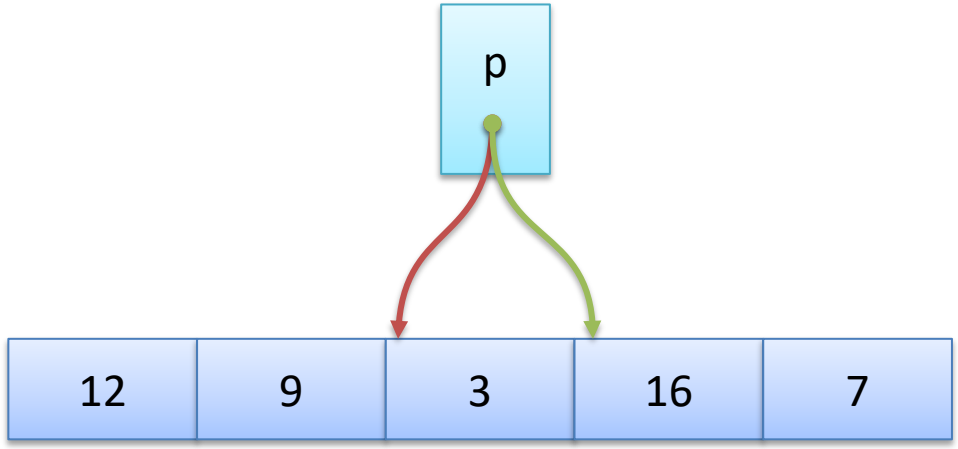
```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



Pointer Arithmetic

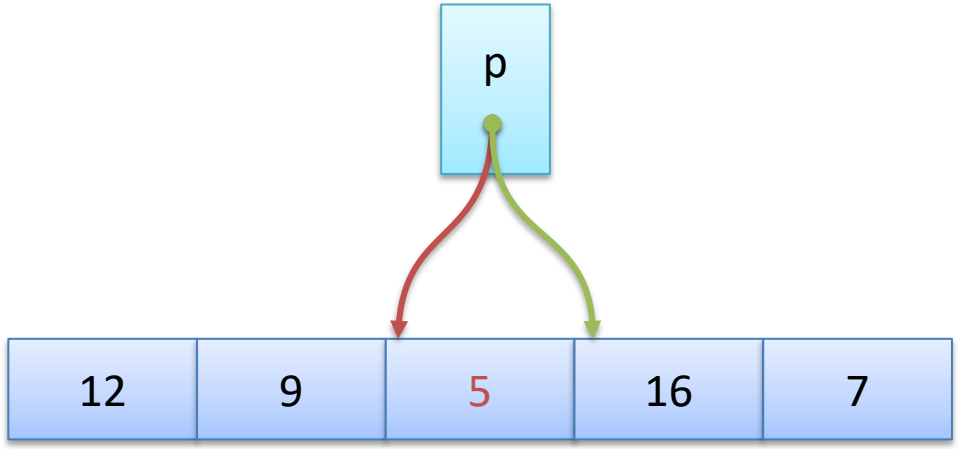



```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



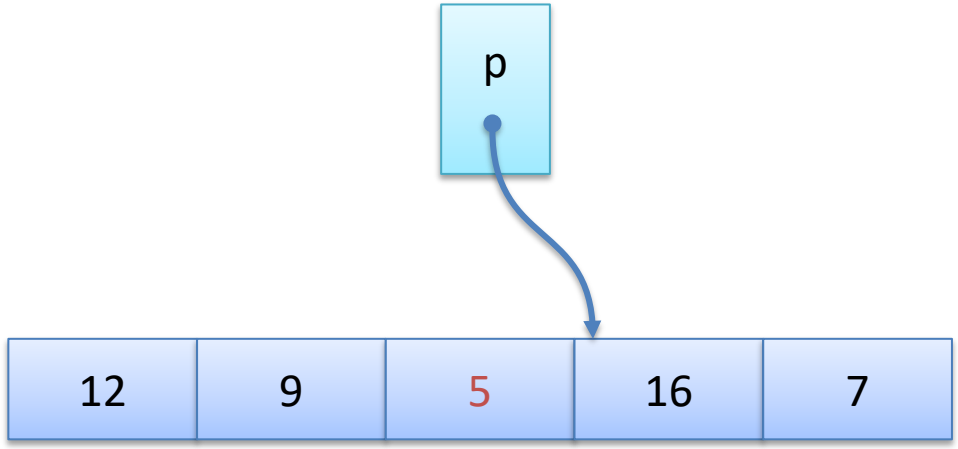
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



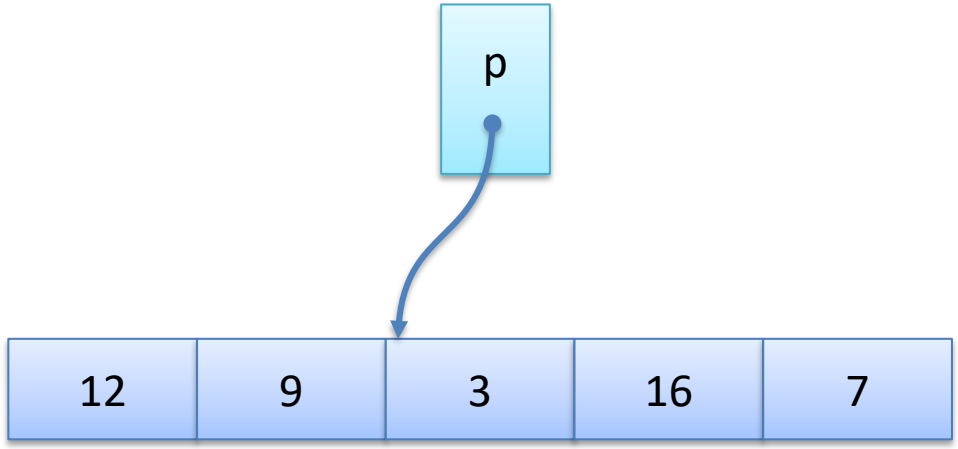
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



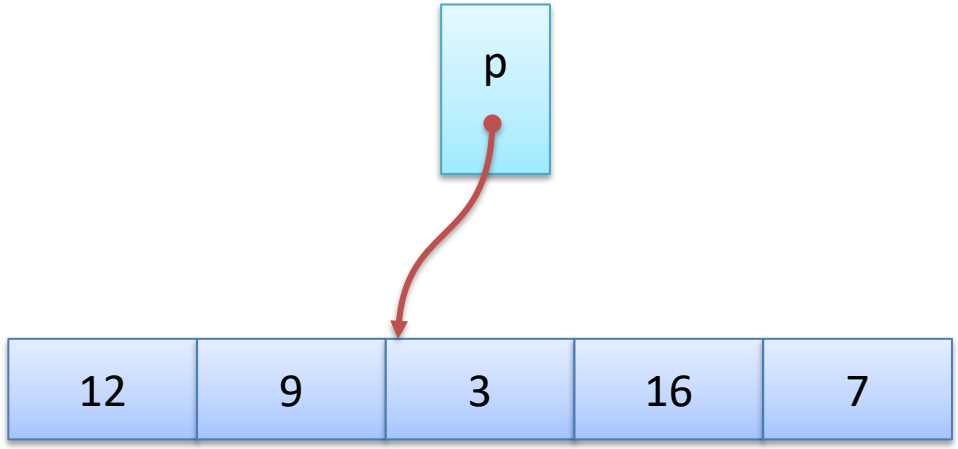
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



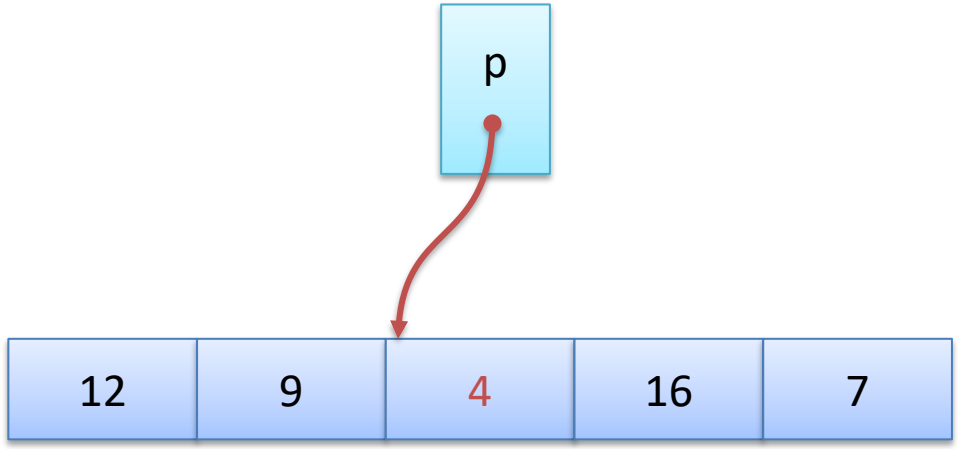
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



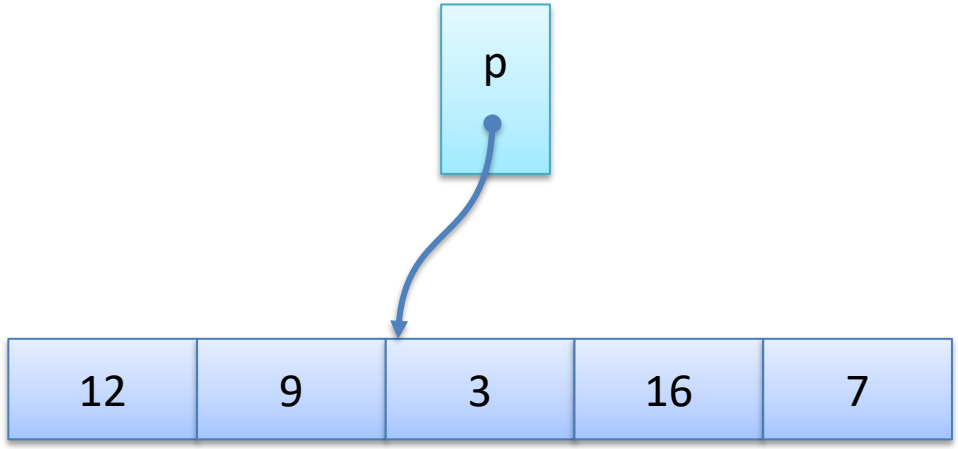
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



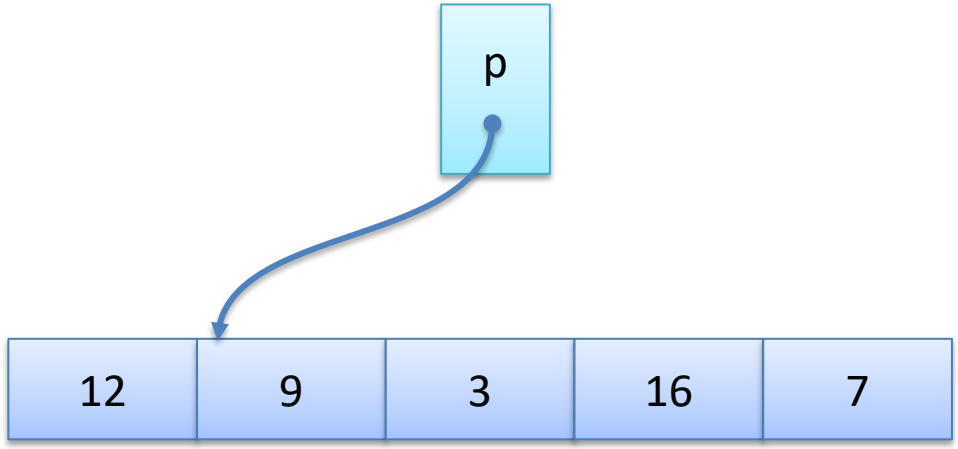
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



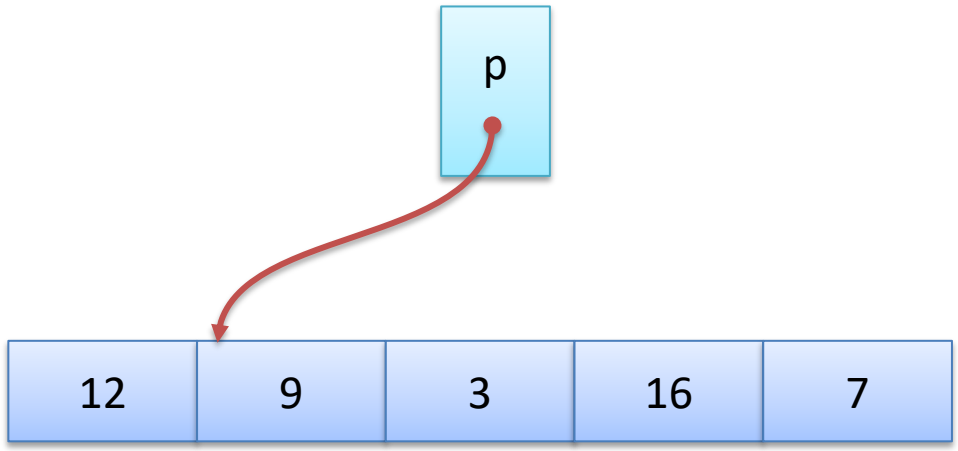
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



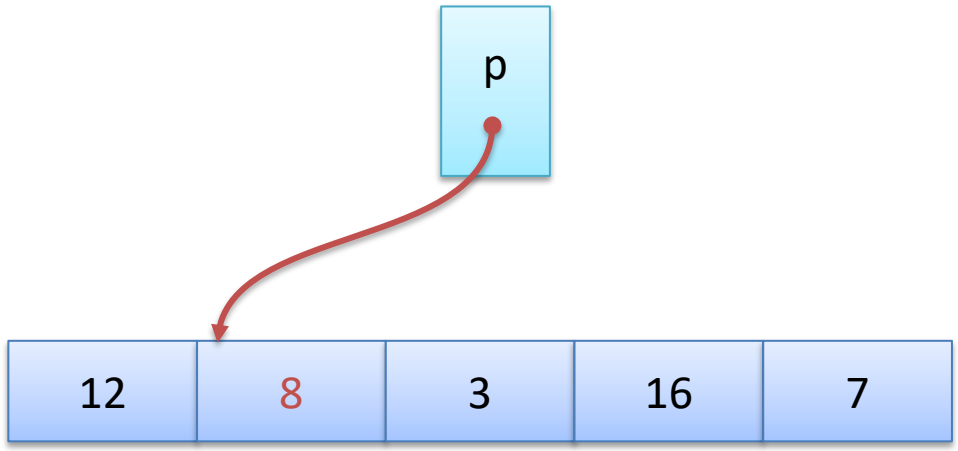
Pointer Arithmetic

```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



Pointer Arithmetic

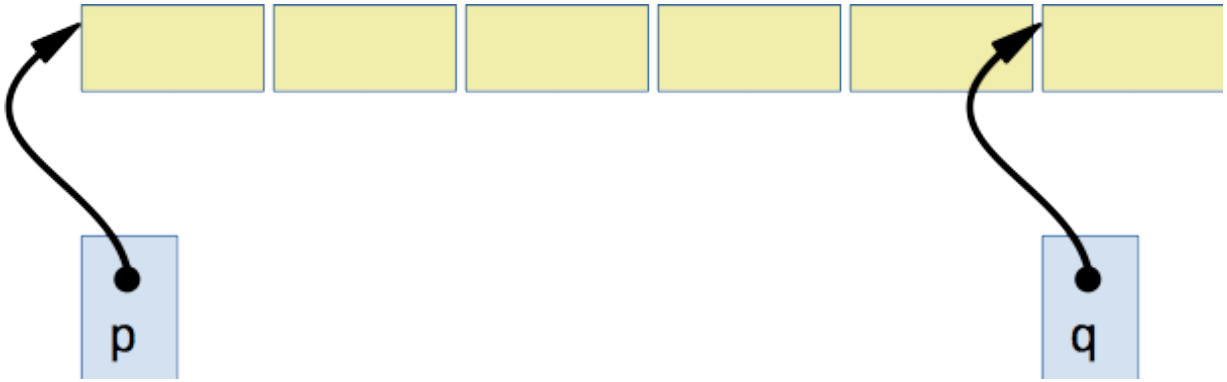
```
p++;
--p;
*--p = 5;
--*p;
*p++ = 5;
(*p)++;
--*--p;
```



Useful Pointer Arithmetic

- You can add and subtract pointers and numbers

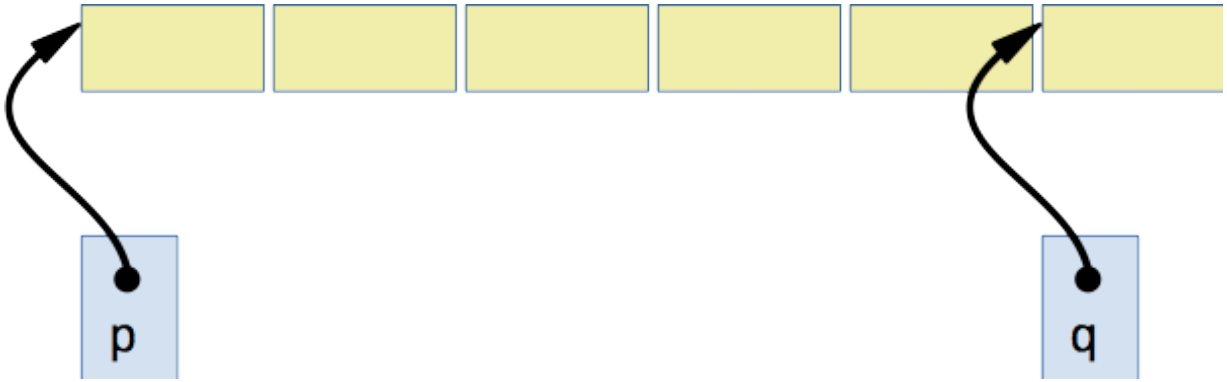
pointer + number → pointer	These each evaluate to a pointer to a nearby value.
number + pointer → pointer	
pointer – number → pointer	
pointer – pointer → number	This evaluates to the number of values between two nearby pointers.
number – pointer → nonsense	Not all operations are meaningful.



Useful Pointer Arithmetic

- You can compare pointers

pointer == pointer	pointer != pointer
pointer < pointer	pointer > pointer
pointer <= pointer	pointer >= pointer



Alternatives in Array Indexing

- We have some choices in how we can process and iterate over arrays.
- Plain-old array syntax:

```
int a[] = { 1, 4, 9, 16, 25 };
int len = sizeof( a ) / sizeof( a[ 0 ] );
```

```
int sum = 0;
for ( int i = 0; i < len; i++ )
  sum += a[ i ];
```

Alternatives in Array Indexing

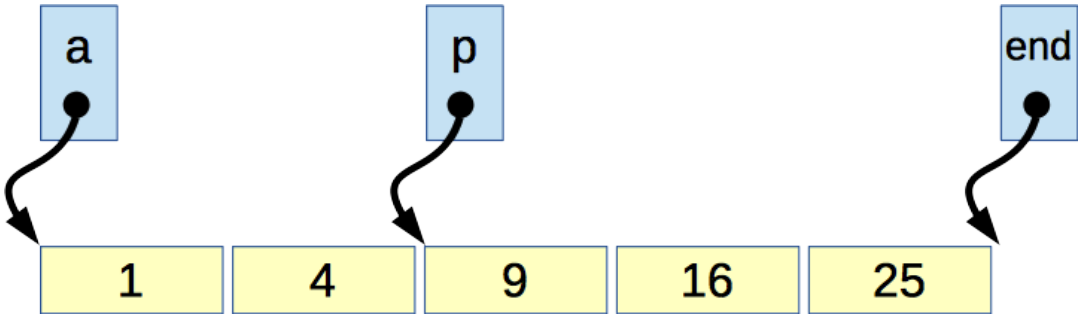
- Or, with a pointer visiting each element.

```
int a[] = { 1, 4, 9, 16, 25 };
int len = sizeof( a ) / sizeof( a[ 0 ] );

int *end = a + len;

int sum = 0;
for ( int *p = a; p < end; p++ )
  sum += *p;
```

Pointer right past the end of the array. This is OK, as long as we don't dereference this pointer.

Iterate from the start of the array, stopping when we pass its last element.

Less to do here. This could give you a little performance improvement.

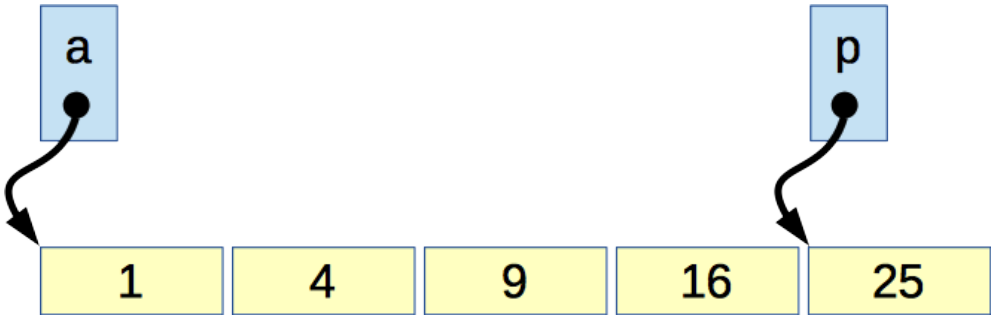


Alternatives in Array Indexing

- Or, we can work backward, from the end toward the start.

```
int a[] = { 1, 4, 9, 16, 25 };
int len = sizeof( a ) / sizeof( a[ 0 ] );
```

```
for ( int *p = a + len - 1; p >= a; p-- )
  sum += *p;
```



Alternatives in Array Indexing

- One more example, measuring string length:

```
int stringLen1( char const *str )
{
  int i = 0;
  while ( str[ i ] != '\0' )
    i++;

  return i;
}
```

```
int stringLen2( char const *str )
{
  char const *p = str;
  while ( *p )
    p++;

  return p - str;
}
```

Here's the interesting part. The difference in pointers tells us how far away p got from the start.

Making Arrays

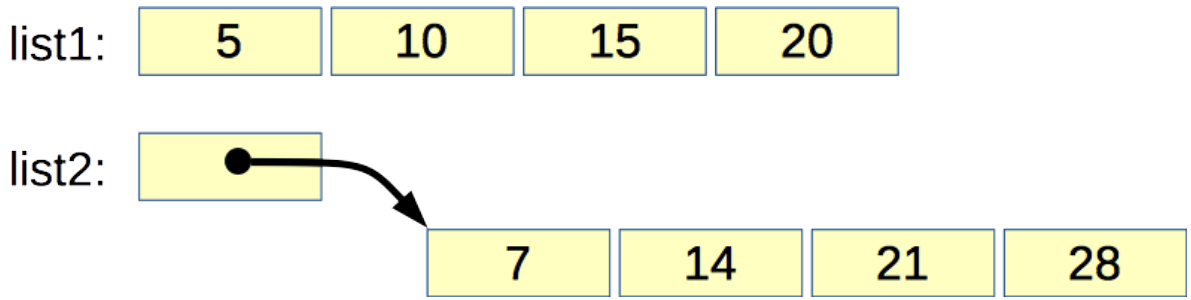
- There are two ways to make a new array.

```
// list1 is a new array.
int list1[] = { 5, 10, 15, 20 };

// list2 is a pointer.
int *list2 = (int []){ 7, 14, 21, 28 };
```

This is the first time we've used this syntax

It makes a new array and gives you a pointer to it.

- We get different things in memory.



Literal Strings

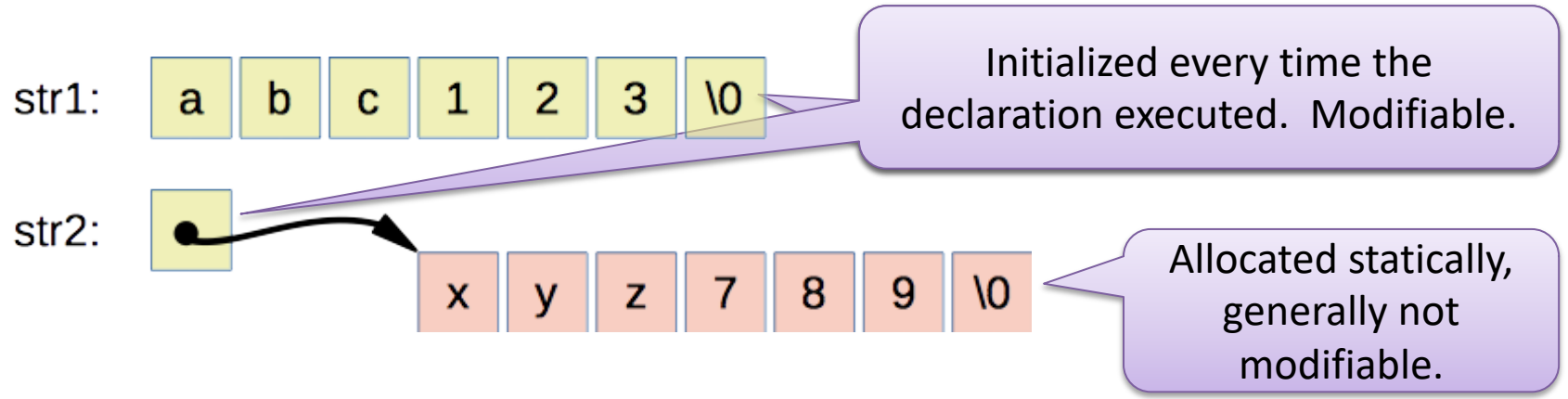
- Literal strings are a special syntax for making an array.
- The constant “abc123” evaluates to a char pointer.
- When the compiler sees a string literal like “abc123”
  - It allocates an array for the string literal (with a null character at the end)
  - And uses the address of this array as the value of the literal
- You can even evaluate things like:
  - “abc123”[2]

String Creation Alternatives

- This gives us some options in how to make a string variable:

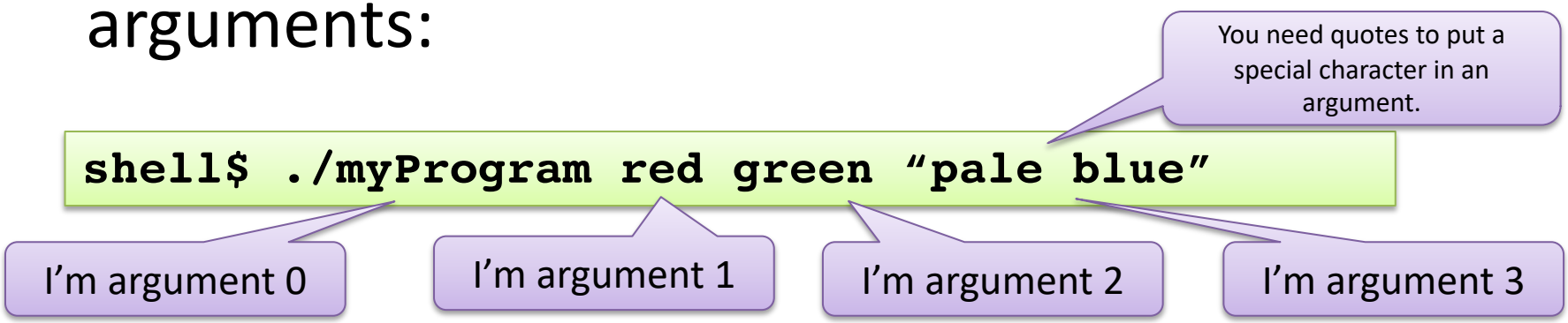
```
// Make me a new array containing this.
char str1[] = "abc123";

// Make this string and just give a pointer
char *str2 = "xyz789";
```

- This allocates and initializes memory differently:



Command Line Arguments

- You can run a C program with command-line arguments:



- These are stored as an array of char pointers, each one pointing to one of the arguments:

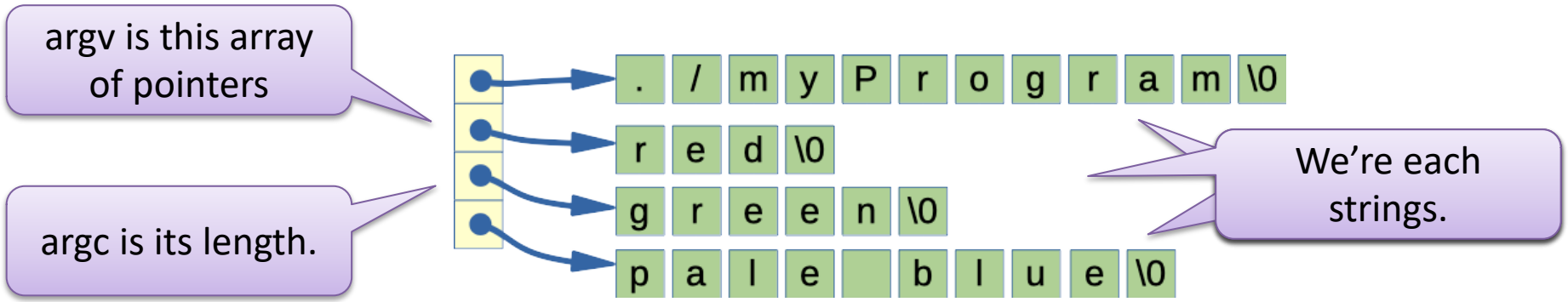
Command Line Arguments

- To see these arguments, you can define main with two parameters:

```
int main( int argc, char *argv[] ) …
```

Number of arguments.

Array of char pointers, one for each argument



Command Line Arguments

- Say, we wanted to just report the arguments:

```
int main( int argc, char *argv[] )
{
  printf( “%d arguments\n", argc );

  for ( int i = 0; i < argc; i++ )
    printf( "    %s\n", argv[ i ] );
}
```

- More typically, we would use the arguments to change how the program operates.