CSc 352

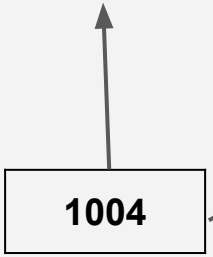
C Programming
Pointers and Arrays

Benjamin Dicken

Pointers

A pointer is a numeric value representing the address of a location memory

```
char x = 190;
char * xp = &x;
```



Address	Values
....	....
1001	105
1002	17
1003	32
1004	190
1005	147
1006	0
1007	100
....	....

C Pointer

- An address to a location memory
- You have access to the number
- Can do *math* with that number
- For static / hard-coded values: compiler assigns pointer
- For dynamic values: malloc assigns values

Java / Py Reference

A reference is sorta like a pointer, however:

- No math on the pointer
- Less control over pointer address, dereference, etc.

Pointers

Address	Values
. . . .	. . . .
0x100000000001	0
0x100000000002	0
. . . .	0
0x100000000010	0
. . . .	0
0x100000000018	0
. . . .	0
0x100000000026	0
0x100000000027	0
. . . .	. . . .

Activity

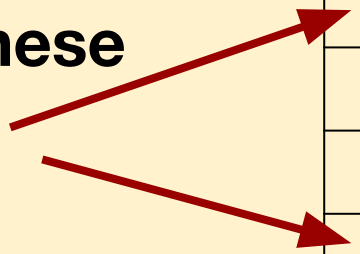
Pointers

How big are these number?

(8 bits? 16 bits? 32 bits? 48 bits? 64 bits?)

Why is it that size?

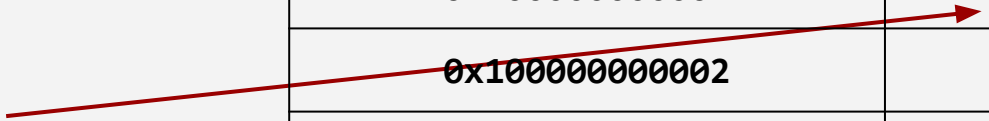
https://en.wikipedia.org/wiki/X86-64#Canonical_form_addresses



Address	Values
. . . .	. . . .
0x100000000001	0
0x100000000002	0
. . . .	0
0x100000000010	0
. . . .	0
0x100000000018	0
. . . .	0
0x100000000026	0
0x100000000027	0
. . . .	. . . .

Pointers

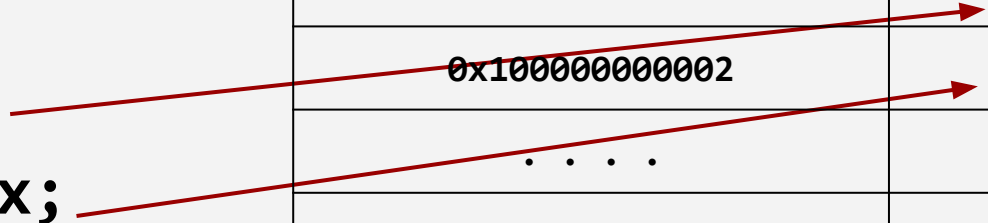
```
char x = 190;
```



Address	Values
. . . .	. . . .
0x100000000001	190
0x100000000002	0
. . . .	0
0x100000000010	0
. . . .	0
0x100000000018	0
. . . .	0
0x100000000026	0
0x100000000027	0
. . . .	. . . .

Pointers

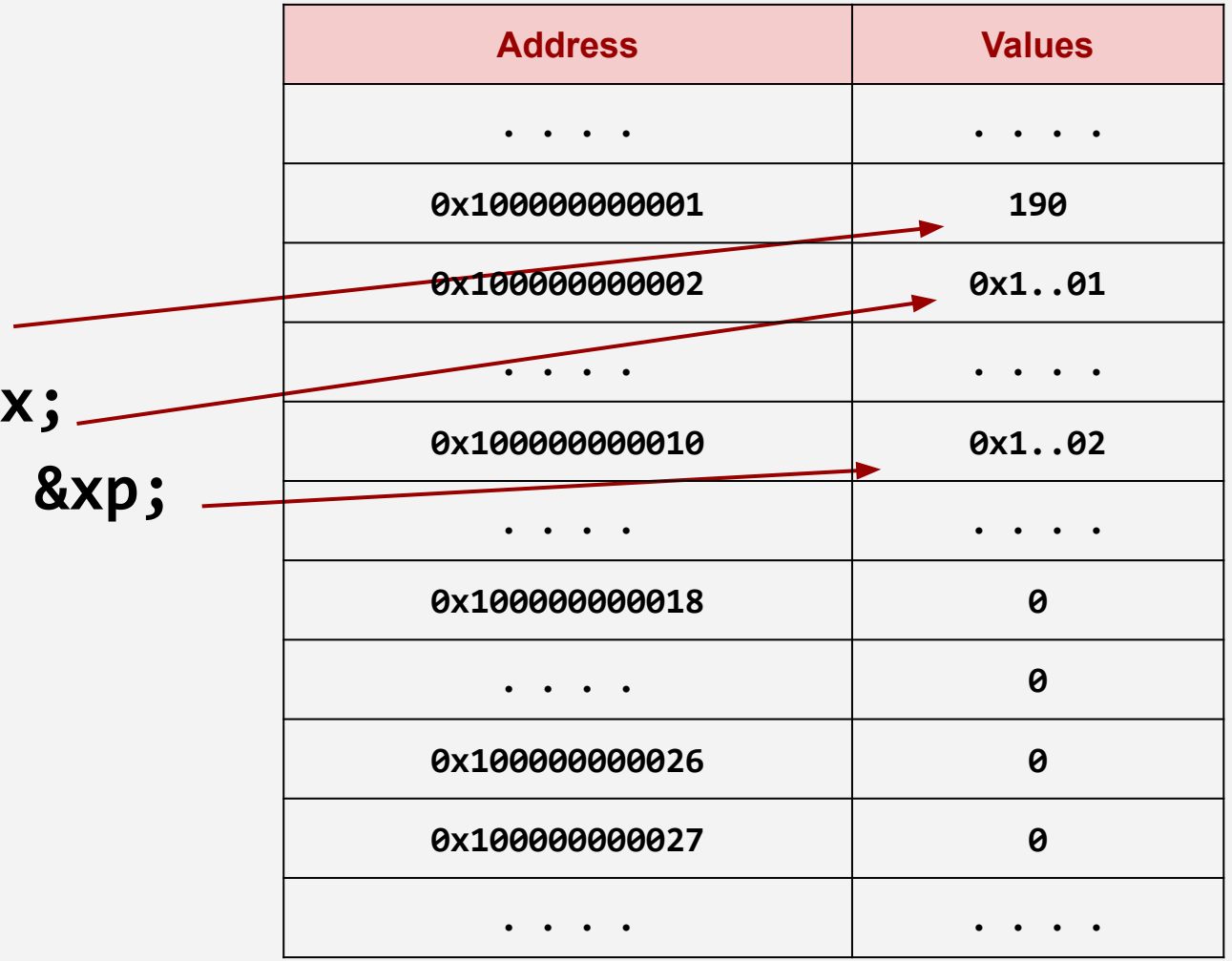
```
char x = 190;
char * xp = &x;
```



Address	Values
. . . .	. . . .
0x100000000001	190
0x100000000002	0x1..01
. . . .	. . . .
0x100000000010	0
. . . .	0
0x100000000018	0
. . . .	0
0x100000000026	0
0x100000000027	0
. . . .	. . . .

Pointers

```
char x = 190;
char * xp = &x;
char ** xpp = &xp;
```



Address	Values
. . . .	. . . .
0x100000000001	190
0x100000000002	0x1..01
. . . .	. . . .
0x100000000010	0x1..02
. . . .	. . . .
0x100000000018	0
. . . .	0
0x100000000026	0
0x100000000027	0
. . . .	. . . .

Pointer-related operators (unary, prefix)

***** dereferences a pointer (gives the values that the pointer points to)

If **x** is a pointer to an int, then ***x** is the int itself

One is the opposite of the other

& gets the address of a value

If **x** is an integer, **&x** is the address of that integer in memory

`*(&p)` is equivalent to **`p`**

What will print?

```c
#include <stdio.h>

int main() {
   int x = 50;
   int * z = &x;
   printf("%d\n", *z);
   return 0;
}
```

What will print?

```c
#include <stdio.h>

int main() {
   int x = 50;
   int ** z = &&x;
   printf("%d\n", **z);
   return 0;
}
```

What will print?

```
int * something(int a,
                int * b) {
  int c = 40;
  a = 20;
  *b = 30;
  int * d = &c;
  return d;
}
```

```
int main() {
  int x = 100;
  int y = 200;
  int * z = something(y, &y);
  printf("%d\n", x);
  printf("%d\n", y);
  printf("%d\n", *z);
  return 0;
}
```

What will print?

```
int * something(int a,
                int * b) {
  int c = 40;
  a = 20;
  *b = 30;
  int * d = &c;
  return d;
}
```

```
int main() {
  int x = 100;
  int y = 200;
  int * z = something(y, &y);
  printf("%d\n", x);
  printf("%d\n", y);
  printf("%d\n", *z);
  return 0;
}
```

What will print?

```c
#include <stdio.h>
void something_else() {
  long r = 200;
  long e = 300;
  printf("%d, %d\n", e, r);
}
int * something() {
  int c = 100;
  int * d = &c;
  return d;
}
```

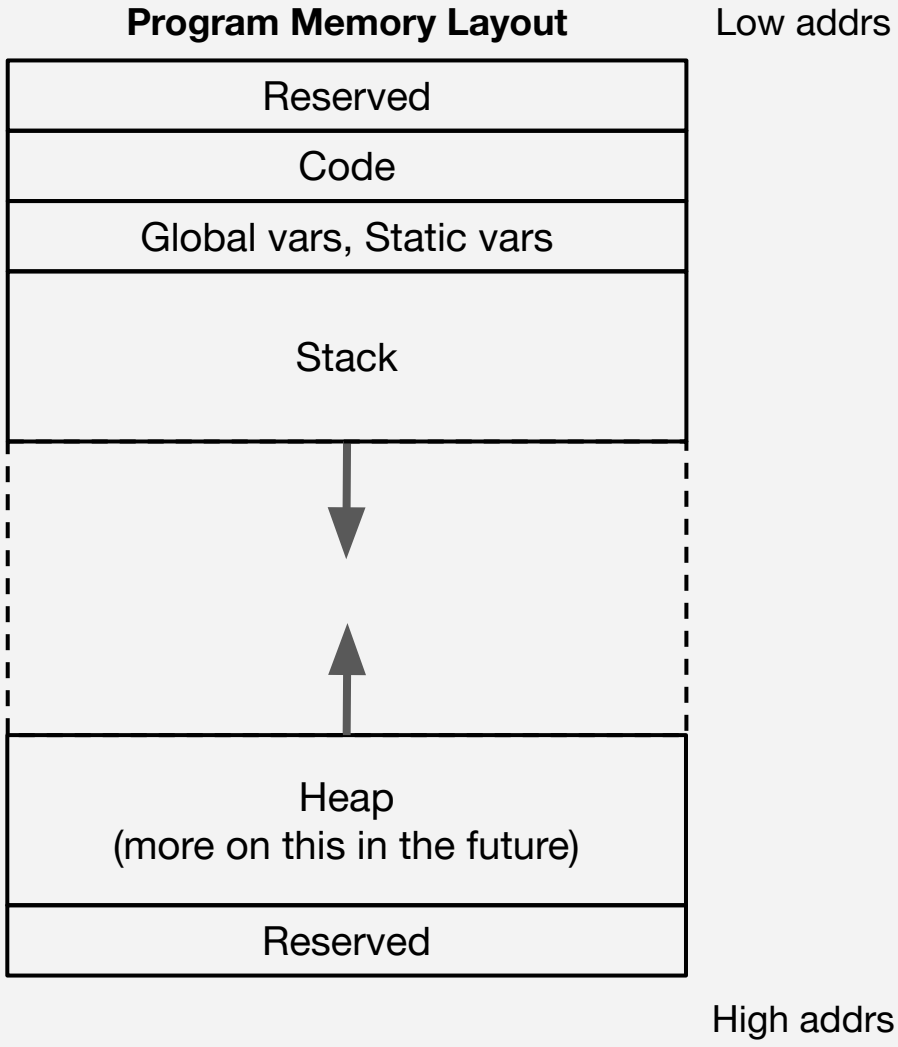
```c
int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```

```c
#include <stdio.h>

void something_else() {
  long r = 200;
  long e = 300;
  printf("%d, %d\n", e, r);
}

int * something() {
  int c = 100;
  int * d = &c;
  return d;
}

int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```

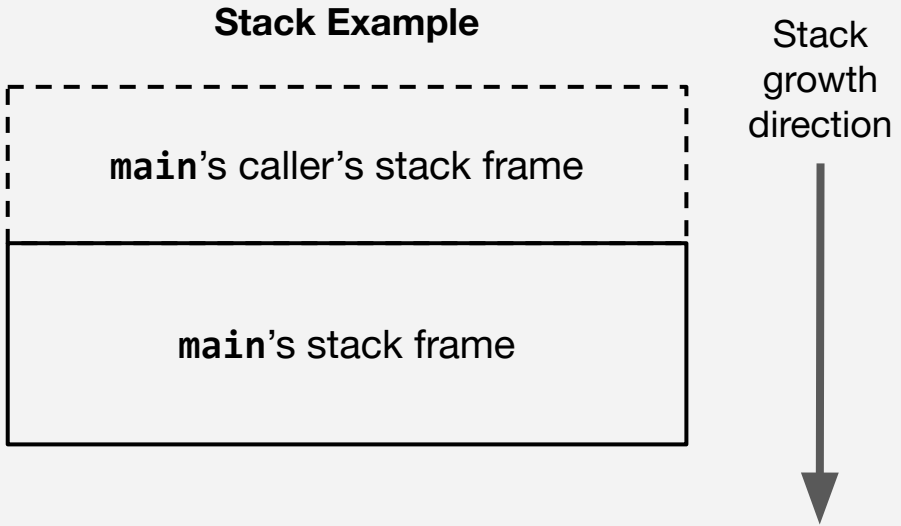


```c
#include <stdio.h>

void something_else() {
  long r = 200;
  long e = 300;
  printf("%d, %d\n", e, r);
}

int * something() {
  int c = 100;
  int * d = &c;
  return d;
}

int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```

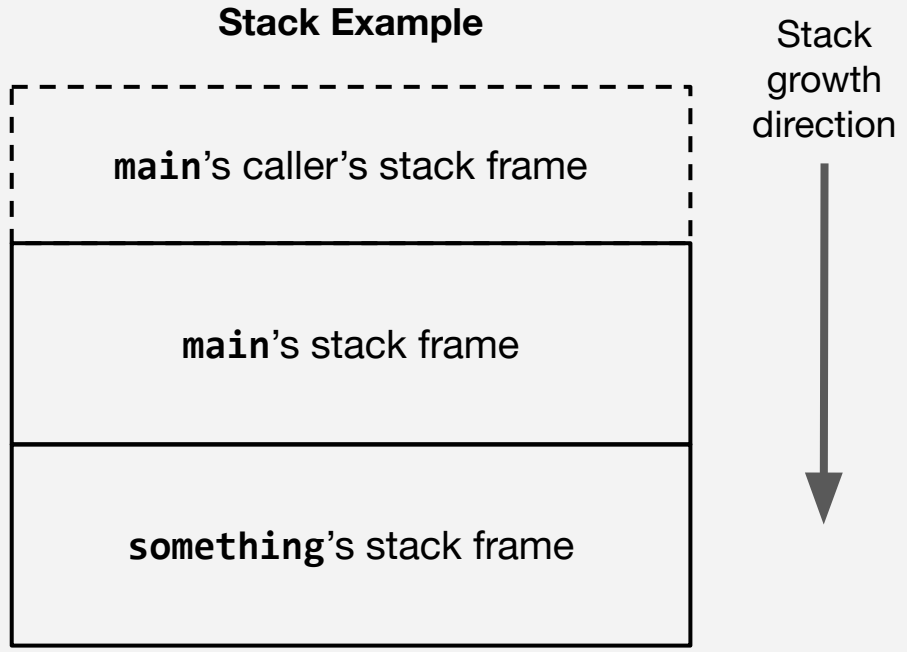


```c
#include <stdio.h>

void something_else() {
  long r = 200;
  long e = 300;
  printf("%d, %d\n", e, r);
}

int * something() {
  int c = 100;
  int * d = &c;
  return d;
}

int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```

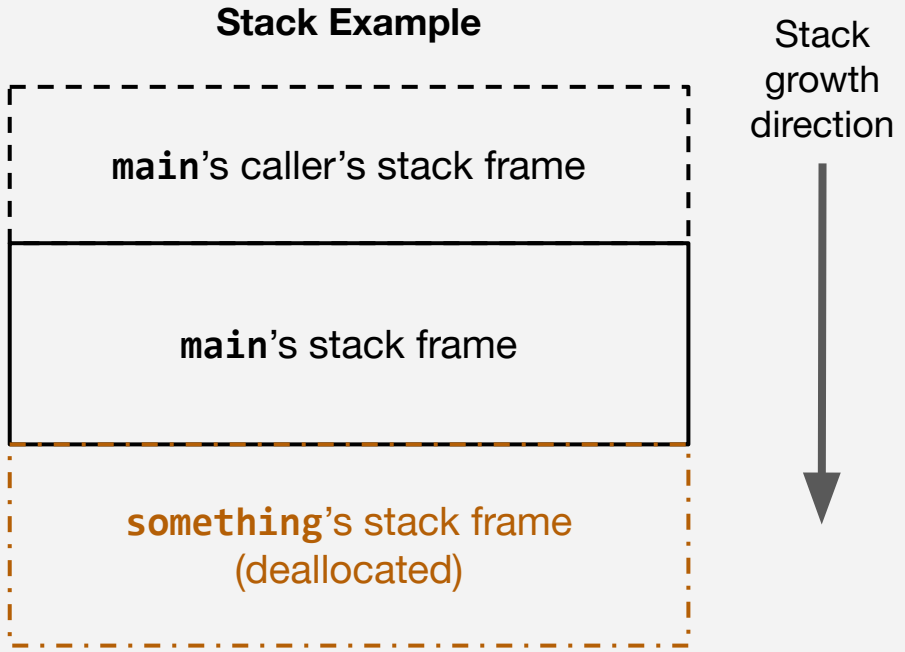


```c
#include <stdio.h>

void something_else() {
  long r = 200;
  long e = 300;
  printf("%d, %d\n", e, r);
}

int * something() {
  int c = 100;
  int * d = &c;
  return d;
}

int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```

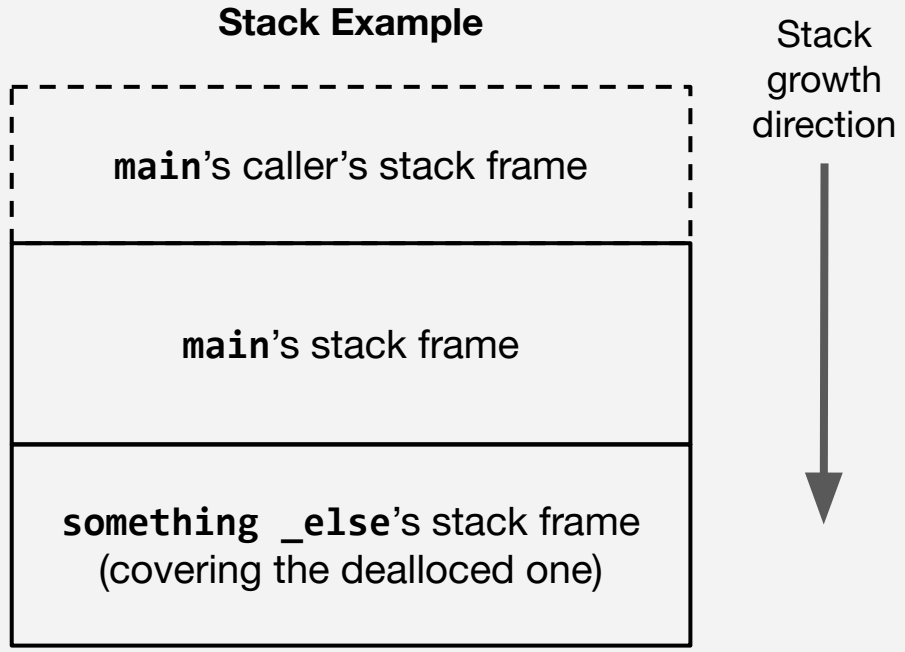


```c
#include <stdio.h>

void something_else() {
  long r = 200;
  long e = 300;
  printf("%d, %d\n", e, r);
}

int * something() {
  int c = 100;
  int * d = &c;
  return d;
}

int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```

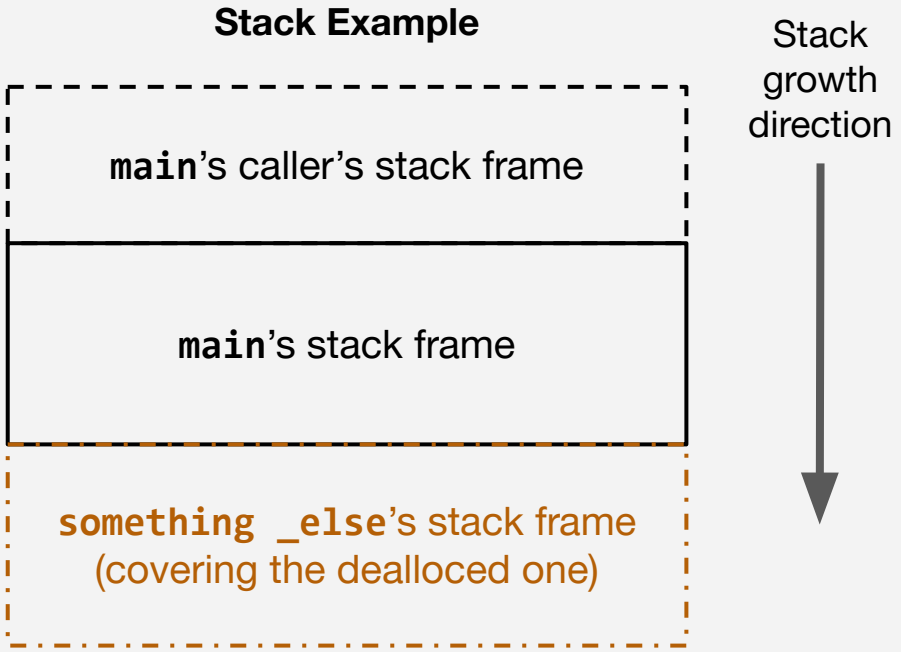


```c
#include <stdio.h>

void something_else() {
  long r = 200;
  long e = 300;
  printf("%d, %d\n", e, r);
}

int * something() {
  int c = 100;
  int * d = &c;
  return d;
}

int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```



Arrays

In C, an array can be thought of as a pointer to a chunk of memory. More specifically, a chunk of memory that is sequential, with a size that is **N * T** where **N** is the number of slots in the array (length) and **T** is the size of the type of the array.

If **char = 1** byte and **int = 4** bytes, then:

```
// name is a ptr to 5 contiguous bytes
char name [5] = "zach";
// numbers is a ptr to 28 contiguous bytes
int numbers[7] = {1, 2, 3, 4, 5, 6, 7};
```

Arrays

```c
// name is a ptr to 5 contiguous bytes
char name [5] = "zach";
// numbers is a ptr to 28 contiguous bytes
int numbers[7] = {1, 2, 3, 4, 5, 6, 7};
int * numbers_2 = numbers;

printf("%p %p %p\n", name, numbers, numbers_2);
printf("%ld %ld\n", sizeof(name), sizeof(numbers));
```

```c
int numbers[7] = {50, 100, 150, 200, 250, 200, 250};
// Code A
for (int i = 0 ; i < 7; i++) {
  printf("element %d is: %d\n", i, numbers[i]);
}
// Code B
for (int i = 0 ; i < 7; i++) {
  printf("element %d is: %d\n", i, *(numbers + i) );
}
// Code C
for (int i = 0 ; i < 7; i++) {
  printf("element %d is: %d\n", i, *numbers++ );
}
```

What is the difference between these codes?

Arrays

- Can do math on pointers!
- Especially useful when dealing with arrays, iterations, offsets

```
char x[4] = {'z', 'r', 't', 'v'};
long y[4] = {100, 200, 300, 400};
char * x2 = x + 1;
long * y2 = y + 2;
printf("%c and %ld", *x2, *y2);
```