A parameter passing example

```
void triple1(int x) { x = x * 3; }
void triple2(int *x) { *x = *x * 3; }
int a[] = {10, 7};
main() {
    triple1(a[0]); /* What is being passed? */
    printf("%d\n", a[0]); /* What is printed? */
    triple2(a); /* What is being passed? */
    printf("%d\n", a[0]); /* What is printed? */
```

- Be sure to understand *why* these results occur.
  - Hint: draw the memory storage – including storage *duration*

Analogous example, re pointers

- First, recall that pointers are values too … then:

```
void repoint1(int *p) { p = p + 1; }
void repoint2(int **p) { *p = *p + 1; }
int a[] = {10, 7};
int *ap = a;
main() {
    repoint1(ap); /* What is being passed? */
    printf("%d\n", *ap); /* What is printed? */
    repoint2(&ap); /* What is being passed? */
    printf("%d\n", *ap); /* What is printed? */
```

`sizeof`

- A unary operator – computes the size, in bytes, of any object or type
  - Usage: `sizeof object` or `sizeof(type)`
    - If `x` is an `int`, `sizeof x == sizeof(int)` is true
  - Especially useful to find the sizes of structures – later
- Works for arrays too – total bytes in whole array
  - Sometimes can use to find an array's length:
    `int size = sizeof x / sizeof x[i];`
- Actually, type of result is `size_t`
  - An unsigned integer defined in `<stddef.h>`
  - Also `ptrdiff_t` – result type of pointer subtraction

2 ways to allocate memory

- Static memory allocation – done at compile-time
  - `int x; double a[5];` /* space for 1 int, 5 doubles */
  - Both size and *type* are clearly specified ahead of time – `x` can only hold `int` values, `a` only `double`s
- Dynamic memory allocation – during execution
  - Must use library functions like `malloc`
    - Allocates specific amount of memory, returns `void *`
      `ip = (int *)malloc(sizeof(int));`
    - So must cast to appropriate pointer type – then use as always
    - Note: `malloc` returns `NULL` if memory is not available
  - To free up dynamic memory: `free(ip);`

Self-Quiz – Pointers & memory

- Say
```
int *a, *b;
a = (int *)malloc(sizeof(int));
b = (int *)malloc(sizeof(int));
*a = 5; *b = 17;
```
  - What does this mean?
  - What are (all) the results of: `a = b;` in this case?
  - What code would swap the values stored at `a` and `b`?
  - What would happen if we tried this: `b = 17;` ?
  - How about this: `printf("b is 0x%x", b);` ?

Returning pointers from functions

- Okay if points to dynamically allocated (or external) storage:

```
int *goodPtr(void) {
    int *p = (int *)malloc(sizeof(int));
    *p = 4;
    return p;      }
```

- Big mistake if points to local storage (inc. parameter values):

```
int *danglingPtr(void) {
    int x = 8;
    int *p = &x;
    return p;          }
```

  - `p` is a dangling pointer – as memory for `x` is erased and/or reused

Multi-dimensional and pointer arrays, and *pointers to pointers*

- Multi-dimensional arrays – arrays of arrays
  - `int x[5][3];` /* allocates memory for 15 ints */
  - Actually, 5 arrays, each able to store 3 integers
- Arrays of pointers
  - `int *p[5];` /* allocates memory for 5 pointers */
    - `for (i=0; i<5; i++) p[i] = x[i];` /* x as above */
    - Now p can be used as an alias for x
- Pointers to pointers
  - `int **px = x;` /* points to first array in x */
  - `px++;` /* moves pointer to next *array* */

First Exam
Friday, October 16