Arrays and Pointer. Part 1

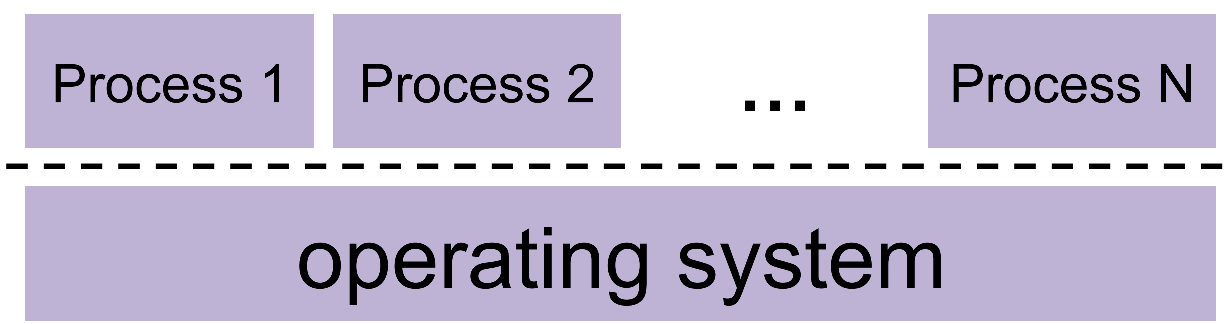
2019 Spring

Lecture Outline

- C's Memory Model (refresher)
- Pointers (refresher)
- Arrays

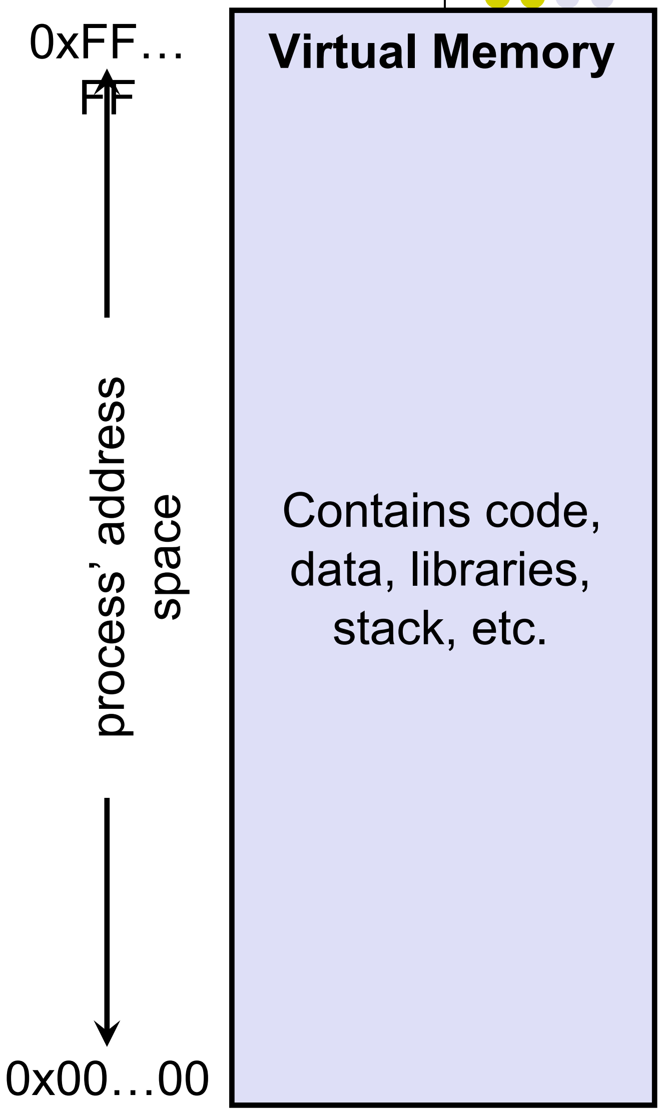
OS and Processes

- The OS lets you run multiple applications at once
  - An application runs within an OS “process”
  - The OS timeslices each CPU between runnable processes
    - This happens *very quickly*: ~100 times per second



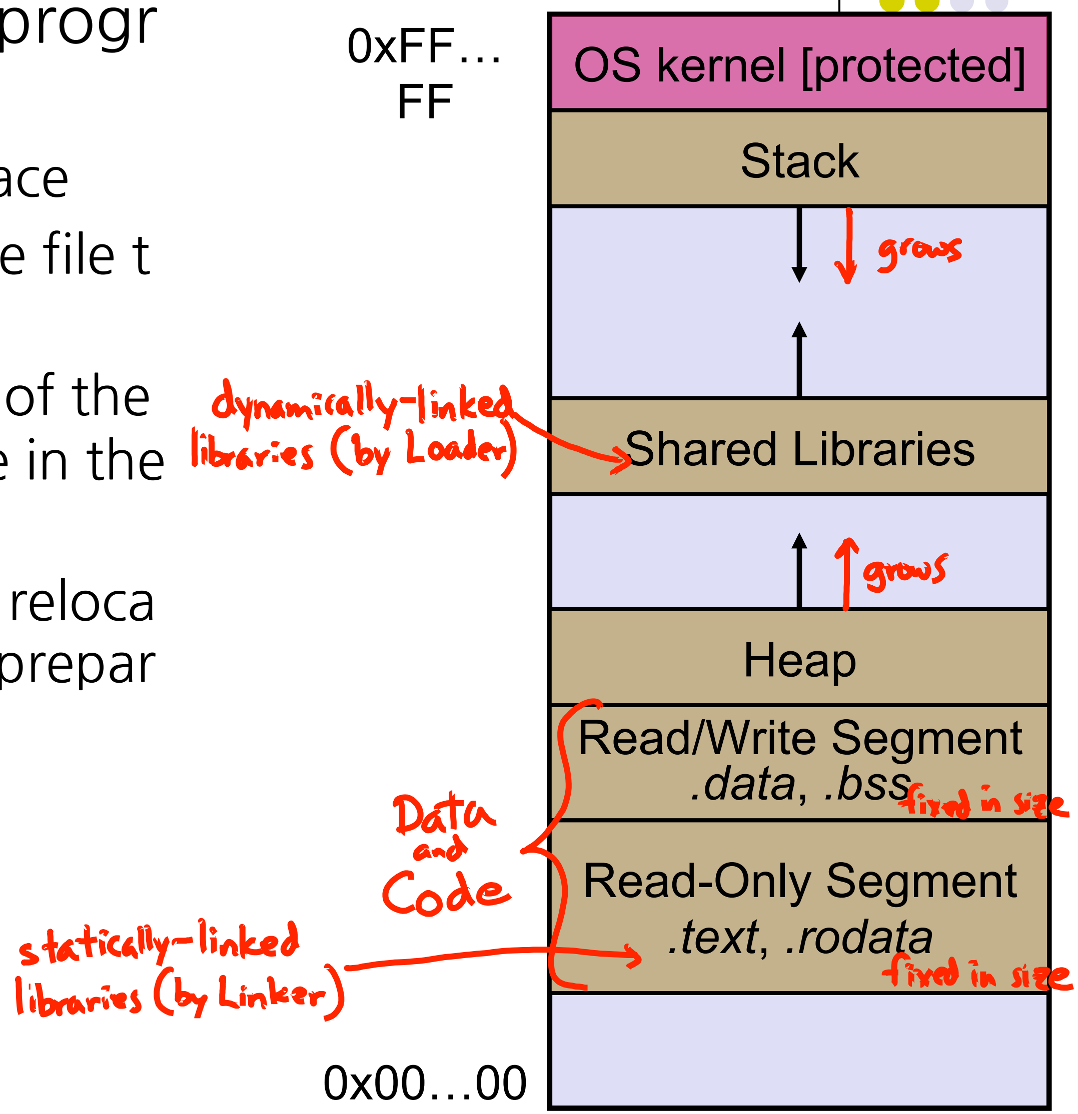
Processes and Virtual Memory

- The OS gives each process the illusion of its own private memory
  - Called the process' address space
  - Contains the process' virtual memory, visible only to it (via translation)
  - 2^{64} bytes on a 64-bit machine



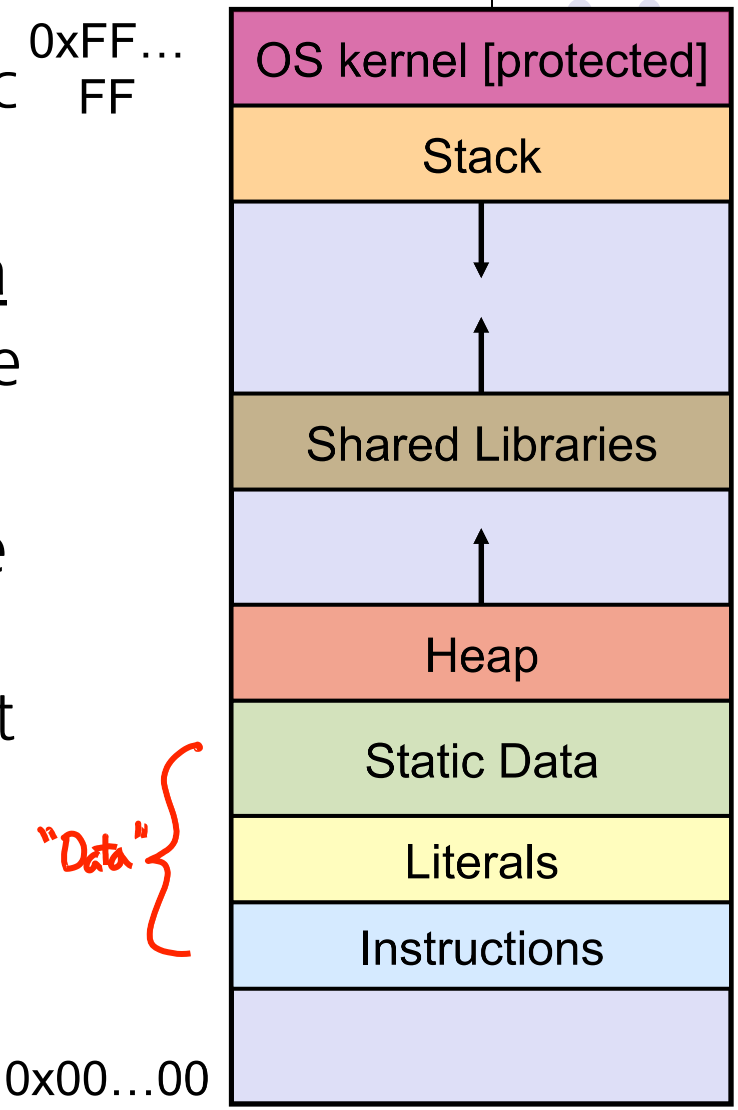
Loading

- When the OS loads a program it:
  1) Creates an address space
  2) Inspects the executable file to see what's in it
  3) (Lazily) copies regions of the file into the right place in the address space
  4) Does any final linking, relocation, or other needed preparation



Memory Management

- *Local* variables on the <u>Stack</u>
  - Allocated and freed via calling conventions (`push`, `pop`, `mov`)
- *Global* and *static* variables in <u>Data</u>
  - Allocated/freed when the process starts/exits
- *Dynamically-allocated* data on the <u>Heap</u>
  - `malloc()` to request; `free()` to free, otherwise memory leak



Stack in Action (1/3)

Note: arrow points to *next* instruction to be executed (like in `gdb`).

stack.c

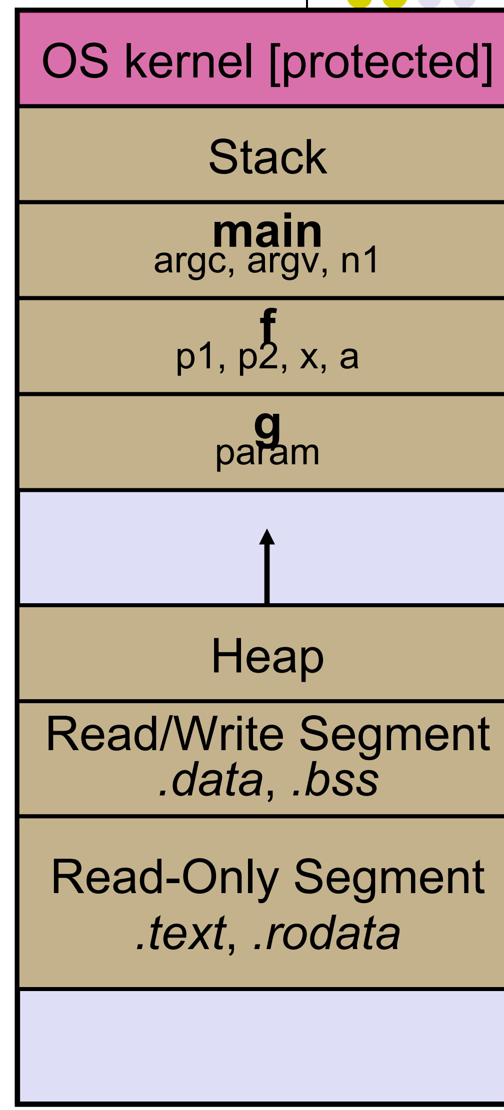
```c
#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
  int n1 = f(3,-5);
  n1 = g(n1);
}

int f(int p1, int p2) {
  int x;
  int a[3];
  ...
  x = g(a[2]);
  return x;
}

int g(int param) {
  return param * 2;
}
```



Stack in Action (2/3)

Note: arrow points to *next* instruction to be executed (like in `gdb`).

stack.c

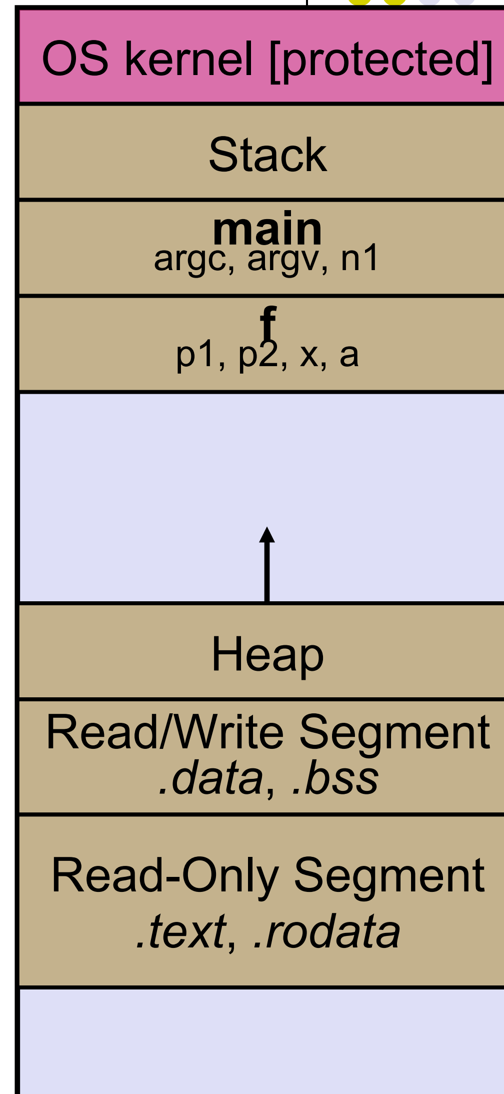
```c
#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
  int n1 = f(3,-5);
  n1 = g(n1);
}

int f(int p1, int p2) {
  int x;
  int a[3];
  ...
  x = g(a[2]);
  return x;
}

int g(int param) {
  return param * 2;
}
```



Stack in Action (3/3)

Note: arrow points to *next* instruction to be executed (like in `gdb`).

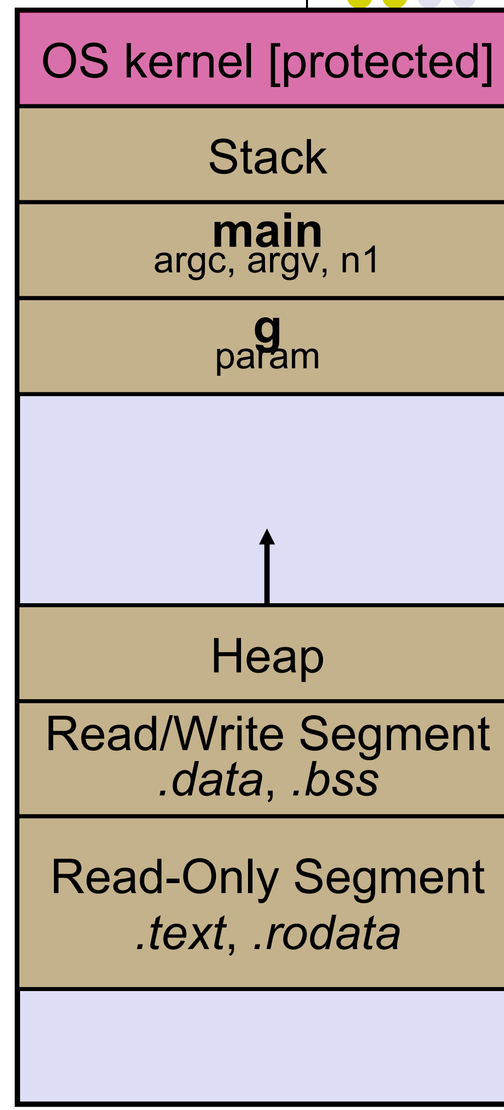
```
#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
  int n1 = f(3,-5);
  n1 = g(n1);
}

int f(int p1, int p2) {
  int x;
  int a[3];
  ...
  x = g(a[2]);
  return x;
}

int g(int param) {
  return param * 2;
}
```



Lecture Outline

- C's Memory Model (refresher)
- Pointers (refresher)
- Arrays

Pointers

- Variables that store addresses
  - It points to somewhere in the process' virtual address space
  - `&foo` produces the virtual address of `foo`
- Generic definition: `type* name;` or `type *name;`
  - Recommended to not define multiple pointers on same line: `int *p1, p2;` not the same as `int *p1, *p2;`
  - Instead, use:

```
int *p1;
int *p2;
```

- *Dereference* a pointer using the unary `*` operator
  - Access the memory referred to by a pointer

Pointer Example

pointy.c

```
#include <stdio.h>
#include <stdint.h>

int main(int argc, char** argv) {
  int x = 351;
  int* p;        // p is a pointer to a int

  p = &x;        // p now contains the addr of x
  printf("&x is %p\n", &x);
  printf(" p is %p\n",  p);
  printf(" x is %d\n",  x);

  *p = 333;     // change value of x
  printf(" x is %d\n",  x);

  return 0;
}
```

Something Curious

- What happens if we run `pointy.c` several times?

```
$ gcc -Wall -std=c11 -o pointy pointy.c
```

Run 1:

```
bash$ ./pointy
&x is 0x7ffff9e28524
 p is 0x7ffff9e28524
 x is 351
 x is 333
```

Run 2:

```
bash$ ./pointy
&x is 0x7fffe847be34
 p is 0x7fffe847be34
 x is 351
 x is 333
```

Run 3:

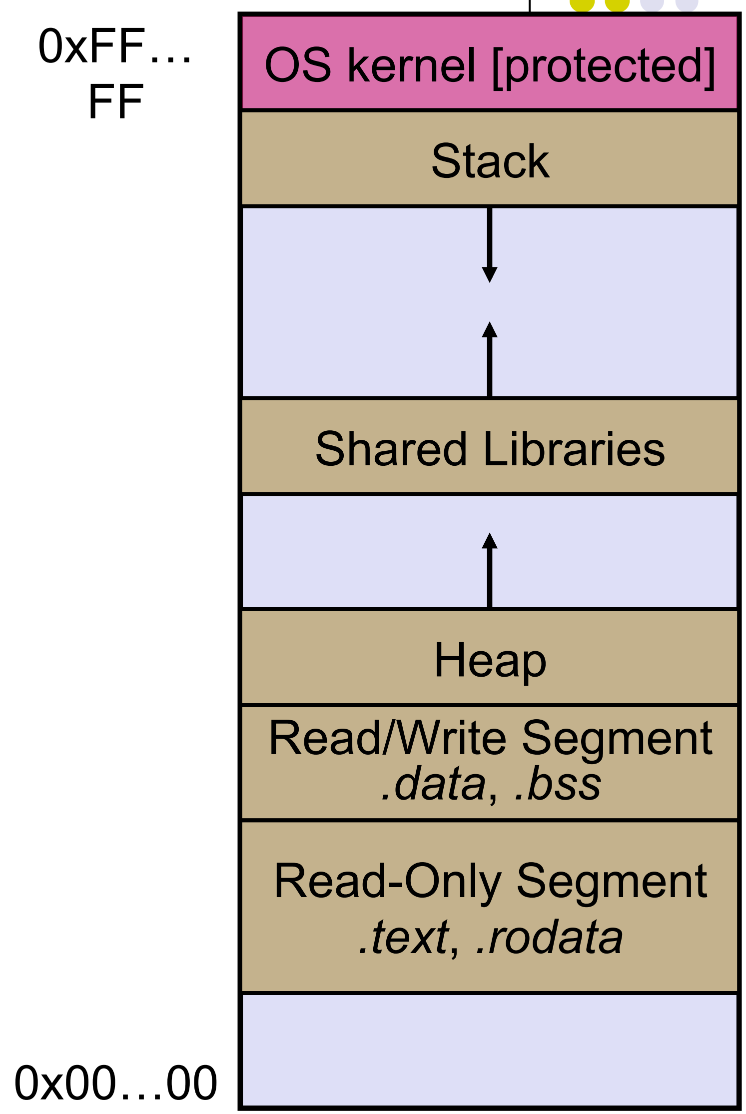
```
bash$ ./pointy
&x is 0x7fffe7b14644
 p is 0x7fffe7b14644
 x is 351
 x is 333
```

Run 4:

```
bash$ ./pointy
&x is 0x7fffff0dfe54
 p is 0x7fffff0dfe54
 x is 351
 x is 333
```

Address Space Layout Randomization

- Linux uses *address space layout randomization* (ASLR) for added security
  - Randomizes:
    - Base of stack
    - Shared library (`mmap`) location
  - Makes Stack-based buffer overflow attacks tougher
  - Makes debugging tougher
  - Can be disabled (`gdb` does this by default); Google if curious



Lecture Outline

- C's Memory Model (refresher)
- Pointers (refresher)
- **Arrays**

Arrays

- <u>Definition</u>: `type name[size]`
  - Allocates `size*sizeof(type)` bytes of *contiguous* memory
  - Normal usage is a **compile-time constant** for `size` (*e.g.* `int scores[175];`)
  - Initially, array values are "garbage"

Size of an array

- Size of an array
  - Not stored anywhere - array does not know its own size!
    - `sizeof(array)` only works in variable scope of array defini tion
  - Recent versions of C allow for variable-length arrays
    - Uncommon and can be considered bad practice [*we won't use*]
      - https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html

```
int n = 175;
int scores[n];  // OK in C99
```

Challenge Question

should malloc instead of using vla's!

- The code snippets both use a variable-length array. What will happen when we compile with C99?

allocated in Static Data (can't change size)

```
int m = 175;
int scores[m];

void foo(int n) {
  ...
}
```

```
int m = 175;

void foo(int n) {
  int scores[n];
  ...
}
```

allocated on the stack (can grow)

A. Compiler Error	Compiler Error
B. Compiler Error	No Error
C. No Error	Compiler Error
D. No Error	No Error
E. We're lost…	

however, you don't want to put large arrays on the stack

Using Arrays (1/2)

- Initialization

```
type name[size] = {val0,…,valN};
```

  - `{}` initialization can *only* be used at time of definition
  - If no size supplied, infers from length of array initializer

Using Arrays (2/2)

- Array name used as identifier for “collection of data”
  - `name[index]` specifies an element of the array and can be us ed as an assignment target or as a value in an expression
  - Array name (by itself) produces the address of the start of the array
    - Cannot be assigned to / changed

```
int primes[6] = {2, 3, 5, 6, 11, 13};
primes[3] = 7;
primes[100] = 0;   // memory smash!
```

Multi-dimensional Arrays

- Generic 2D format:
  `type name[rows][cols] ={{values},…,{values}};`

  - Still allocates a single, contiguous chunk of memory
  - C is *row-major*

```
// a 2-row, 3-column array of doubles
double grid[2][3];

// a 3-row, 5-column array of ints
int matrix[3][5] = {
  {0, 1, 2, 3, 4},
  {0, 2, 4, 6, 8},
  {1, 3, 5, 7, 9}
};
```

Parameters: reference vs. value (1/2)

- There are two fundamental parameter-passing schem es in programming languages

- Call-by-value
  - Parameter is a local variable initialized when the function is cal led and gets a copy of the calling argument; manipulating the parameter only changes copy, *not* the calling argument
  - **C**, **Java**, C++ primitives

Parameters: reference vs. value (2/2)

- Call-by-reference
  - Parameter is an alias for the supplied argument; manipulating the parameter manipulates the calling argument
  - C++ arrays and references (we'll see more later)

Arrays as Parameters

- It's tricky to use arrays as parameters
  - What happens when you use an array name as an argument?
  - Arrays do not know their own size

```
int sumAll(int a[]);  // prototype

int main(int argc, char** argv) {
  int numbers[] = {9, 8, 1, 9, 5};
  int sum = sumAll(numbers);
  return 0;
}

int sumAll(int a[]) {
  int i, sum = 0;
  for (i = 0; i < ...???
}
```

Solution 1: Declare Array Size

```c
int sumAll(int a[5]);  // prototype

int main(int argc, char** argv) {
  int numbers[] = {9, 8, 1, 9, 5};
  int sum = sumAll(numbers);
  printf("sum is: %d\n", sum);
  return 0;
}

int sumAll(int a[5]) {
  int i, sum = 0;
  for (i = 0; i < 5; i++) {
    sum += a[i];
  }
  return sum;
}
```

- Problem: loss of generality/flexibility!

Solution 2: Pass Size as Parameter

```
int sumAll(int a[], int size);  // prototype

int main(int argc, char** argv) {
  int numbers[] = {9, 8, 1, 9, 5};
  int sum = sumAll(numbers, 5);
  printf("sum is: %d\n", sum);
  return 0;
}

int sumAll(int a[], int size) {
  int i, sum = 0;
  for (i = 0; i < size; i++) {
    sum += a[i];
  }
  return sum;
}
```

Returning an Array

- Local variables, including arrays, are allocated on the Stack
  - They "disappear" when a function returns!
  - Can't safely return local arrays from functions
    - Can't return an array as a return value - why not?

```
int* copyArray(int src[], int size) {
  int i, dst[size];    // OK in C99

  for (i = 0; i < size; i++) {
    dst[i] = src[i];
  }

  return dst;  // no compiler error, but wrong!
}
```

buggy_copyarray.c

Solution: Output Parameter

- Create the "returned" array in the caller
  - Pass it as an output parameter to `copyarray()`
    - A pointer parameter that allows the callee to leave values fo r the caller to use
  - Works because arrays are "passed" as pointers
    - "Feels" like call-by-reference, *but it's not*

```
void copyArray(int src[], int dst[], int size) {
  int i;

  for (i = 0; i < size; i++) {
    dst[i] = src[i];
  }
}
```

copyarray.c

Output Parameters

- Output parameters are common in library functions
  - `long int strtol(char* str, char** endptr, int base);`
  - `int sscanf(char* str, char* format, ...);`

```
int   num, i;
char* pEnd, str1 = "333 rocks";
char  str2[10];

// converts "333 rocks" into long -- pEnd is conversion end
num = (int) strtol(str1, &pEnd, 10);

// reads string into arguments based on format string
num = sscanf("3 blind mice", "%d %s", &i, str2);
```

outparam.c

http://www.cplusplus.com/reference/cstdlib/strtol/
http://www.cplusplus.com/reference/cstdio/sscanf/

Questions?