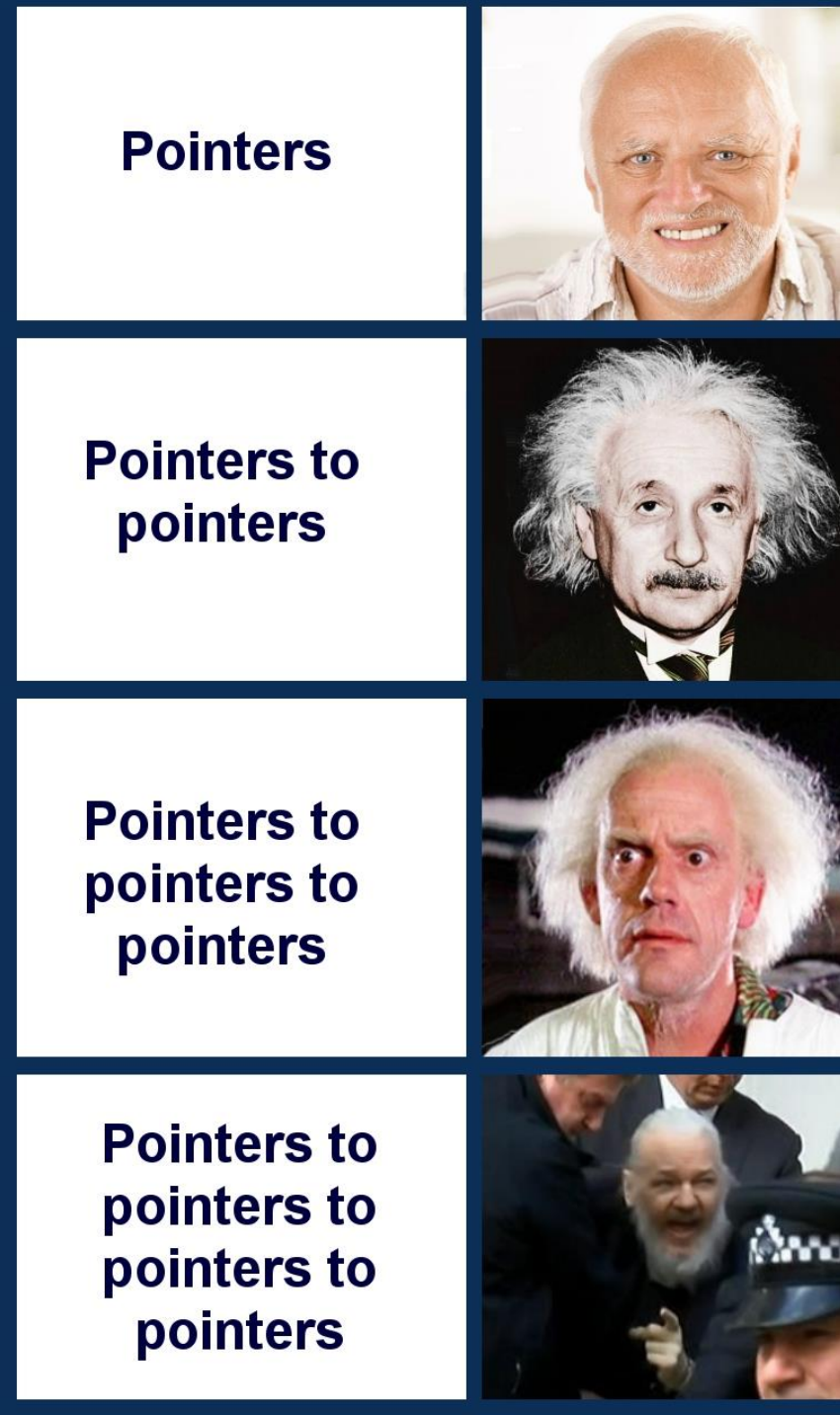
CMPSC 311 - Introduction to Systems Programming

Arrays and Pointers

Professors
Sencun Zhu and Suman Saha
(Slides are mostly by Professor Patrick McDaniel
and Professor Abutalib Aghayev)



Pointers

- `type *name;` `// declare a pointer`
- `type *name = address;` `// declare + initialize a pointer`
- a pointer is a variable that contains a memory address
- Source of confusion: *p vs p – the pointer variable is just p, without *

```c
int main(void) {
  int x = 42;
  int *p;       // int* p; also works

  p = &x;       // p now stores the address of x

  printf("x  is %d\n", x);
  printf("&x is %p\n", &x);
  printf("p  is %p\n", p);

  return 0;
}
```

A stylistic choice

- C gives you flexibility in how you declare pointers
  - one way can lead to visual trouble when declaring multiple pointers on a single line
  - the other way is often preferred

```
int* p1;
int *p2; // preferred
```

```
int* p1, p2;   // bug?; equivalent to int *p1; int p2;
int* p1, * p2; // correct

or

int *p1, *p2;  // correct, preferred (generally)
```

Dereferencing pointers

PennState

- dereference: access the memory referred to by a pointer
  - `*pointer  // dereference a pointer`
  - `*pointer is an alias for the variable pointer points to`
  - `*pointer = value;  // dereference / assign`
  - `pointer-p = pointer-q; // pointer assignment`

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int x = 42;
  int *p;       // p is a pointer to an integer
  p = &x;       // p now stores the address of x

  printf("x  is %d\n", x);
  *p = 99;
  printf("x  is %d\n", x);

  return 0;
}
```

Pointers as function arguments

PennState

- Pointers allow C to emulate pass by reference
  - Enables modifying **out parameters,** efficient passing of **in parameters**

```
void min_max(int array[], int len, int *min, int *max) {
  // find the index of largest value and assign it to max_i
  // find the index of the smallest value and assign it to min_i
  *max = array[max_i];
  *min = array[min_i];
}

int main(void) {
  int x[100];
  int largest, smallest;
  // some code that populates the array x...
  min_max(x, 100, &smallest, &largest);
  printf("smallest = %d, largest = %d\n", smallest, largest);
}
```

Pointers as return values

PennState

- Okay to return passed in pointer (or dynamically allocated memory)

```
int *max(int *a, int *b) {
  if (*a > *b) return a;
  return b;
}

int main(void) {
  int x = 5, y = 6;
  printf("max of %d and %d is %d\n", x, y, *max(&x, &y));
}
```

- NOT OKAY TO RETURN THE ADDRESS OF A LOCAL VARIABLE

```
int *foo(void) {
  int x = 5;
  return &x;
}

int main(void) {
  printf("%d\n", *foo());
}
```

Variable Storage Classes

PennState

- C (and other languages) have several storage classes that are defined by their scopes
  - auto – these are automatically allocated and deallocated variables (local function variables declared on stack)
  - global – globally defined variables that can be accessed anywhere within the program
    - keyword `extern` is used in .c/.h files to indicate a variable defined elsewhere
  - static – a variable that is global to the local file only (static global variable) or to the scope of a function (static local variable)
    - keyword `static` is used to identify variable as local only

```
static int localScopeVariable; // Static variable
extern int GlobalVariable;     // Global variable defined elsewhere
```

More examples: https://overiq.com/c-programming-101/local-global-and-static-variables-in-c/

Rules for initialization

PennState

- In general, static or global variables are given a default value (often zero) and auto storage class variables are indeterminate
  - meaning that the compiler can do anything it wants, which for most compilers is take whatever value is in memory
  - You cannot depend on indeterminate values

> *C89 Specification* : If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate. If an object that has static storage duration is not initialized explicitly, it is initialized implicitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a null pointer constant.

> *C99 Specification* : If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate. If an object that has static storage duration is not initialized explicitly, then:
> — if it has pointer type, it is initialized to a null pointer;
> — if it has arithmetic type, it is initialized to (positive or unsigned) zero;
> — if it is an aggregate, every member is initialized (recursively) according to these rules;
> — if it is a union, the first named member is initialized (recursively) according to these rules.

Variable Storage Class Example

- global: declared outside of any function (without any keyword)
  - accessible from anywhere within the same file
  - accessible in other files via extern keyword

```
 1 #include <stdio.h>
 2
 3 int x = 42;
 4
 5 int triple(void);
 6
 7 int main(void) {
 8   printf("%d\n", x);
 9   printf("%d\n", triple());
10 }
```

main.c

```
1 extern int x;
2
3 int triple(void) {
4   return x * 3;
5 }
```

triple.c

```
neo@ubuntu~$ gcc -Wall main.c triple.c -o foo
neo@ubuntu~$ ./foo
42
126
```

- static: declared outside of any function with static keyword
  - prepending static to line 3 of main.c limits scope of x to main.c; compilation fails

Variable Storage Class Example (cont.)

PennState

- global and static variables:
  - initialized to supplied or default value before program execution begins
  - preserve changes until the end of program execution
- static variables can also appear within a function:
  - limits the scope to the function
  - unlike automatic variables, preserves changes across invocations

```
#include <stdio.h>
void foo(void) {
  int x = 0; static int y = 0;
  x += 1; y += 1;
  printf("x=%d y=%d\n", x, y);
}
int main(void) {
  foo();
  foo();
}
```

foo.c

```
neo@ubuntu~$ gcc -Wall foo.c -o foo
neo@ubuntu~$ ./foo
x=1 y=1
x=1 y=2
```

Function Storage Class Example

PennState

- Storage classes also apply to functions
- By default, functions are global: line 5 in main.c has an implicit extern

```
 1 #include <stdio.h>
 2
 3 int x = 42;
 4
 5 int triple(void);
 6
 7 int main(void) {
 8   printf("%d\n", x);
 9   printf("%d\n", triple());
10 }
```

main.c

```
1 extern int x;
2
3 int triple(void) {
4   return x * 3;
5 }
```

triple.c

```
neo@ubuntu~$ gcc -Wall main.c triple.c -o foo
neo@ubuntu~$ ./foo
42
126
```

- To limit visibility of **`triple`** to file prepend static in line 3; compilation fails
- Functions in C cannot be nested; hence no static function within a function

Arrays

PennState

- type name[size];

```
int scores[100];
```

- example allocates 100 ints worth of memory
  - initially, each array element contains garbage data
- an array does not know its own size
  - `sizeof(scores)` is not reliable; only works in some situations
  - C99 standard allows the array size to be an expression

```
int n=100;
int scores[n];  // OK in C99
```

C Arrays *are zero indexed*!

Initializing and using arrays

PennState

- `type name[size] = {value, value, ..., value};`
  - allocates and array and fills it with supplied values
  - if fewer values are given than the array size, fills rest with 0

```
// Best approach to init all values
int val4[3] = { [0 ... 2] = 1 };
```

- `name[index] = <expression>;`
  - sets the value of an array element

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

Multi-dimensional arrays

- *type name[rows][columns] = {{values}, ..., {values}};*
  - allocates a 2D array and fills it with predefined values

```
// 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} };

grid[0][2] = (double) matrix[2][4];  // which val?
```

Multi-dimensional arrays

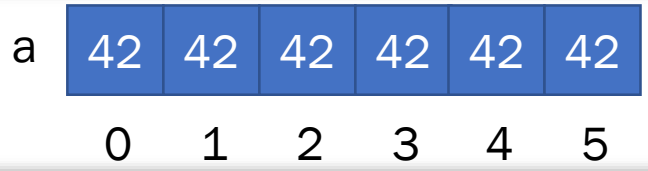
- *type name[rows][columns] = {{values}, ..., {values}};*
  - allocates a 2D array and fills it with predefined values

```
// 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} };

grid[0][2] = (double) matrix[2][4];  // which val?
```

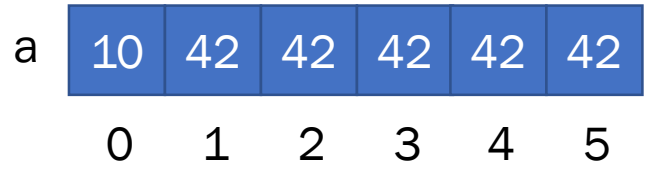
Arrays and pointers

- Array name can be used as a pointer to 0th elem

```
int a[6] = { [0 ... 5] = 42 };
printf("%d\n", *a);              // prints 42
```

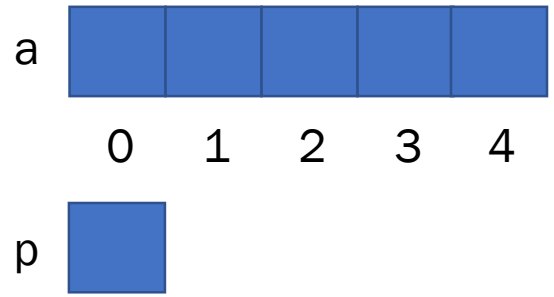



```
*a = 10;
```



- Array name and pointers are different otherwise

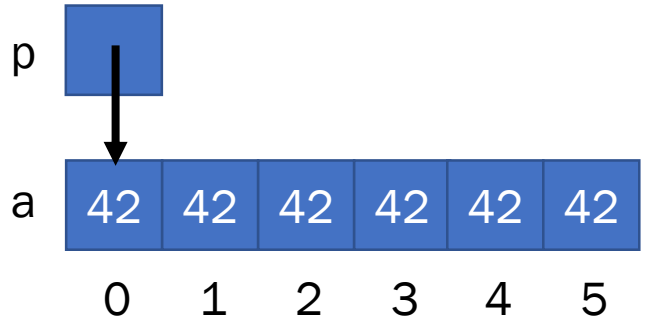
```
int a[5];
int *p; int x = 5;
p = &x; // OK
a = &x; // WILL FAIL TO COMPILE
p++;    // OK
a++;    // WILL FAIL TO COMPILE
```



Pointer Arithmetic

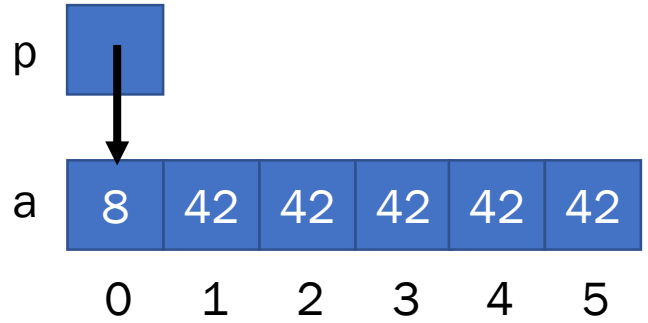
- We can make pointers to point to array elements

```
int a[6] = { [0 ... 5] = 42 };
int *p = &a[0];
```



- We can access array elements through pointers

```
*p = 8;
```

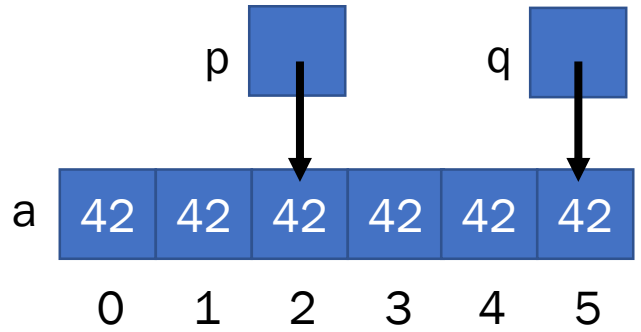


- We can do arithmetic on a pointer that points to array elements
  - Add integer to pointer
  - Subtract integer from a pointer
  - Subtract one pointer from another

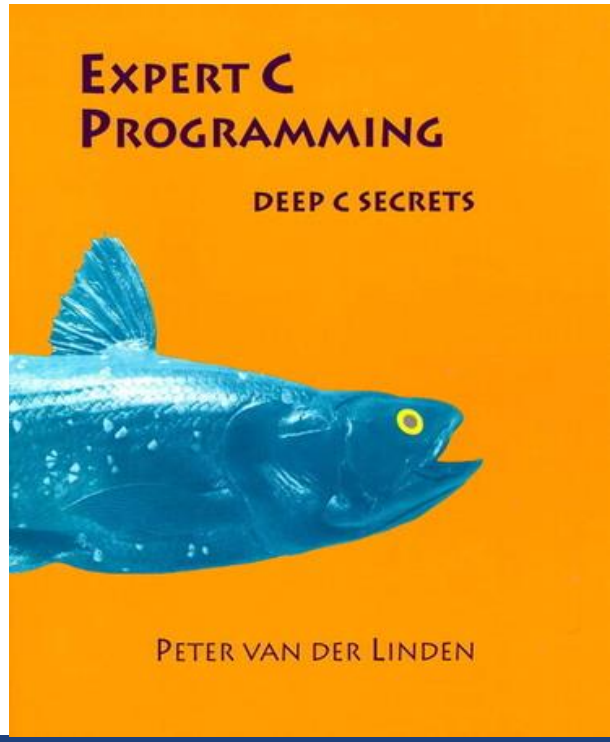
Pointer Arithmetic (cont.)

- Adding integer i to pointer p yields a pointer to the element i places after the one that p points to (skipping subtracting an integer from a pointer – same idea)

```
int a[6] = { [0 ... 5] = 42 };
int *p = &a[2];
int *q = p + 3;
++p;
```



- Using array name as a pointer: a[i] == *(a+i)
  - a[0] == *(a+0) = *a
  - a[5] == *(a+5)
- Fun trick: a[2] == 2[a]
  - try it: int a[5] = {0, 1, 2, 3, 4}; printf("%d %d\n", a[2], 2[a]);



Pointer Arithmetic (cont.)

- In pointer arithmetic, the pointer is incremented based on the size of a single array element

```
#include <stdio.h>

int main(void) {
  int foo[10];
  int *p = foo; // point to the 0-th element
  printf("%p\n", p);
  ++p;      // point to the next element
  printf("%p\n", p);

  char bar[10];
  char *q = bar; // point to the 0-th element
  printf("%p\n", q);
  ++q;      // point to the next element
  printf("%p\n", q);
}
```

```
neo@ubuntu~$ gcc -Wall foo.c -o foo
neo@ubuntu~$ ./foo
0x7ffd05a9fd50
0x7ffd05a9fd54
0x7ffd05a9fd7e
0x7ffd05a9fd7f
```

- Pointer arithmetic applicable only to pointers that point to arrays (undefined otherwise)
- Subtract pointers only if they point to the same array (undefined otherwise)

Pointer Arithmetic (cont.)

PennState

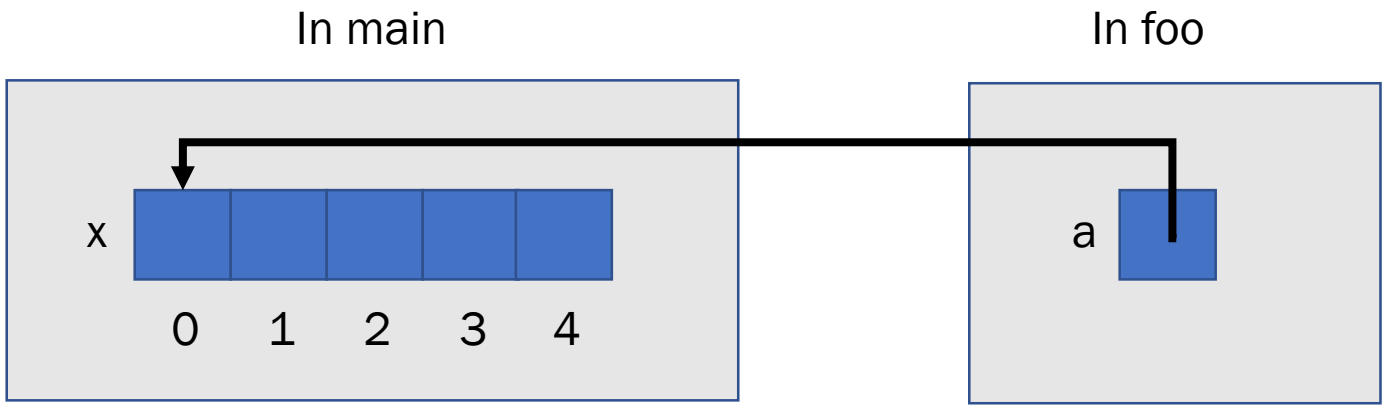
```
#include <stdio.h>

int main(void) {
  int foo[10] = { [0 ... 9] = 42 };
  int sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;

  for (int i = 0; i < 10; ++i)
    sum1 += foo[i];

  for (int *p = &foo[0]; p < &foo[10]; ++p)
    sum2 += *p;

  for (int *p = foo; p < foo + 10; ++p)
    sum3 += *p;

  int *p = foo;
  for (int i = 0; i < 10; ++i)
    sum4 += p[i];

  printf("%d %d %d %d\n", sum1, sum2, sum3, sum4);
}
```

```
neo@ubuntu~$ gcc -Wall sum.c -o sum
neo@ubuntu~$ ./sum
420 420 420 420
```

Arrays as Function Parameters

- When passing an array as an argument to a function, the address of the 0-th element is passed by value – copied to a pointer in the callee:

```
int foo(int a[]) {
  ...
}

int main(void) {
  int x[5];
  ...
  foo(x);
}
```



- Therefore, `int foo(int a[]);` is equivalent to `int foo(int *a);`
- Unlike array variable, array argument inside a function is a real pointer
  - You can increment, decrement, and assign to it
- Arrays are effectively passed by reference – a function doesn't get a copy of array

Passing Array Size to Function

PennState

- Solution 1: declare the array size in the function
  - problem: code isn't very flexible
    - need different functions for different-sized arrays
    - the array size in function signature is ignored by the compiler

```
int sumAll(int a[5]);

int main(void) {
  int numbers[5] = {3, 4, 1, 7, 4};
  int sum = sumAll(numbers);
  return 0;
}

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

Passing Array Size to Function (cont.)

PennState

- Solution 2: pass the size as a parameter

```c
int sumAll(int a[], int size);

int main(void) {
  int numbers[5] = {3, 4, 1, 7, 4};
  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++) {    //  CAN YOU SPOT THE BUG?
    sum += a[i];
  }
  return sum;
}
```

Returning an array

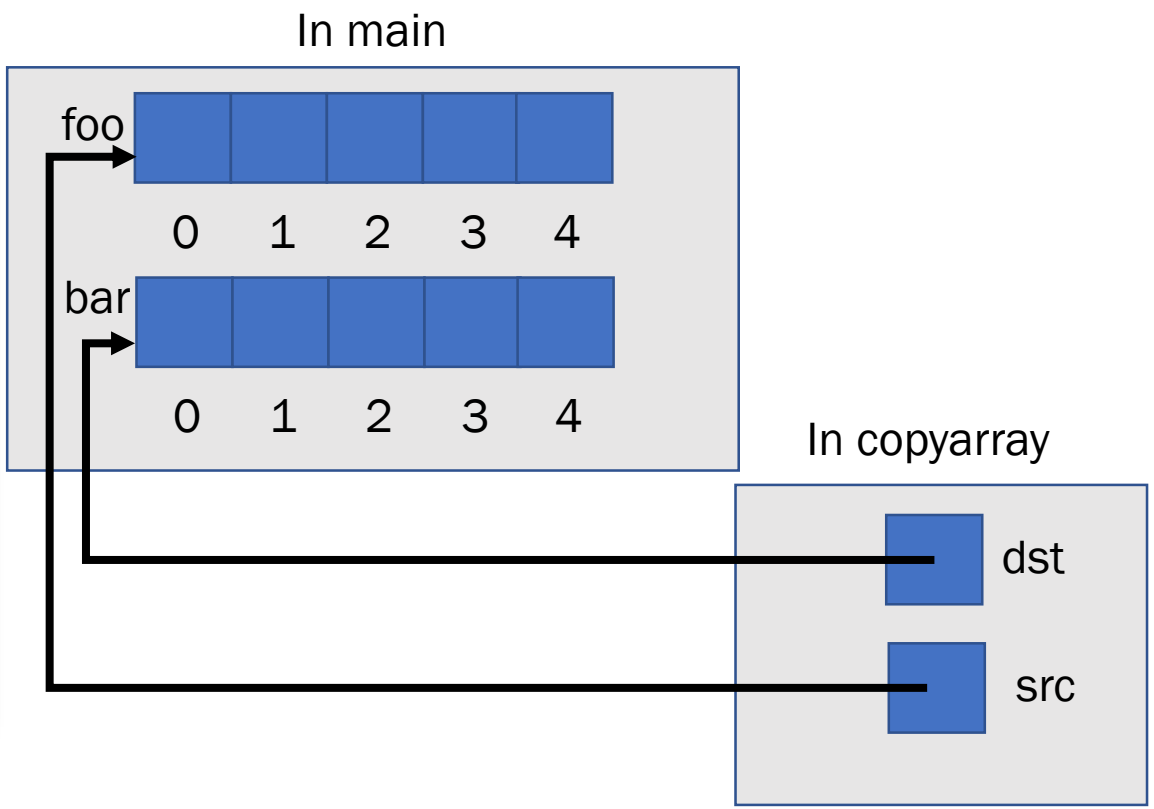
- Local variables, including arrays, are stack allocated
  - they disappear when a function returns
  - therefore, local arrays can't be safely returned from functions

```
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 -- bug
}
```

Solution: an output parameter

- Create the “returned” array in the caller
  - pass it as an output parameter to copyarray
  - works because arrays are effectively passed by reference

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

int main(void) {
  int foo[5] = { [0 ... 4] = 42 };
  int bar[5];
  copyarray(foo, bar, 5);
}
```



Virtual vs Physical Address

PennState

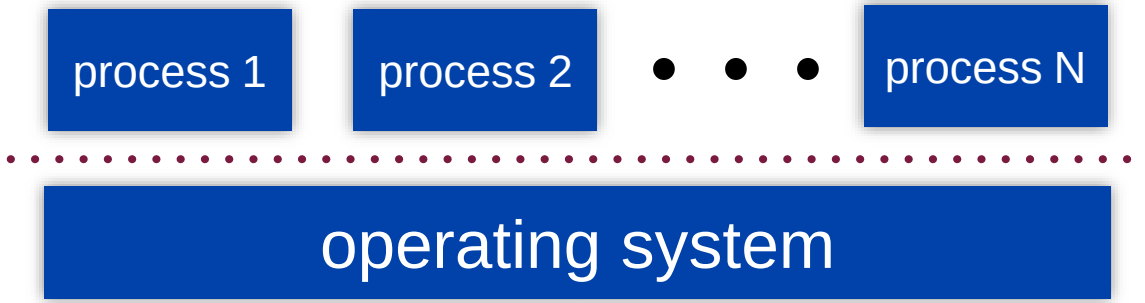
- *&foo* produces the virtual address of *foo*

```c
#include <stdio.h>

int foo(int x) {
  return x+1;
}

int main(void) {
  int x, y;
  int a[2];

  printf("x     is at %p\n", &x);
  printf("y     is at %p\n", &y);
  printf("a[0]  is at %p\n", &a[0]);
  printf("a[1]  is at %p\n", &a[1]);
  printf("foo   is at %p\n", &foo);
  printf("main  is at %p\n", &main);

  return 0;
}
```

```
neo@ubuntu~$ gcc -Wall addr.c -o addr
neo@ubuntu~$ ./addr
x    is at 0x7fff8bfc6bc8
y    is at 0x7fff8bfc6bcc
a[0] is at 0x7fff8bfc6bd0
a[1] is at 0x7fff8bfc6bd4
foo  is at 0x55dd0a63e169
main is at 0x55dd0a63e17c
```

OS and processes (redux)

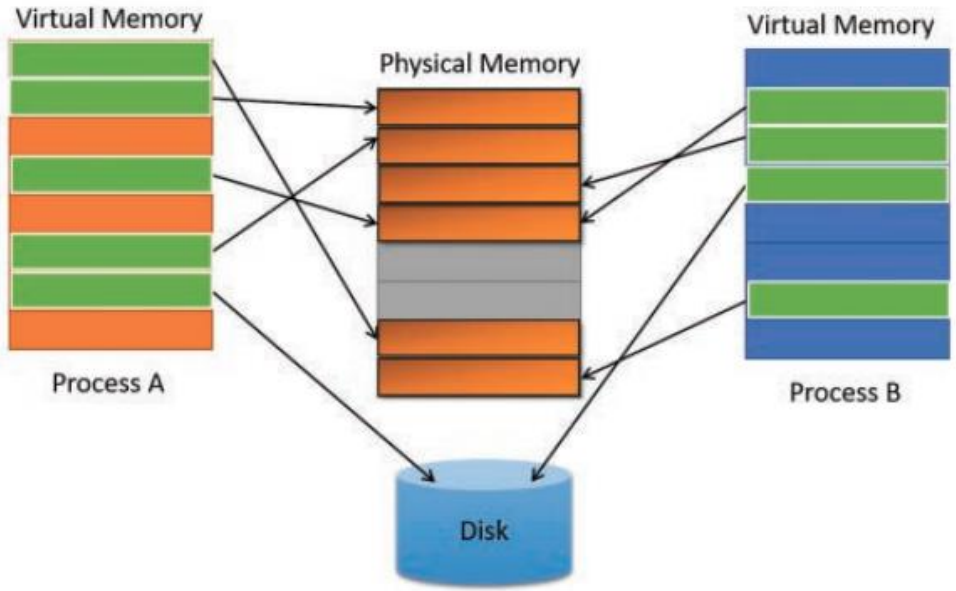
- The OS lets you run multiple applications at once
  - an application runs within an OS “process”
  - the OS “timeslices” each CPU between runnable processes
    - happens very fast; ~100 times per second!

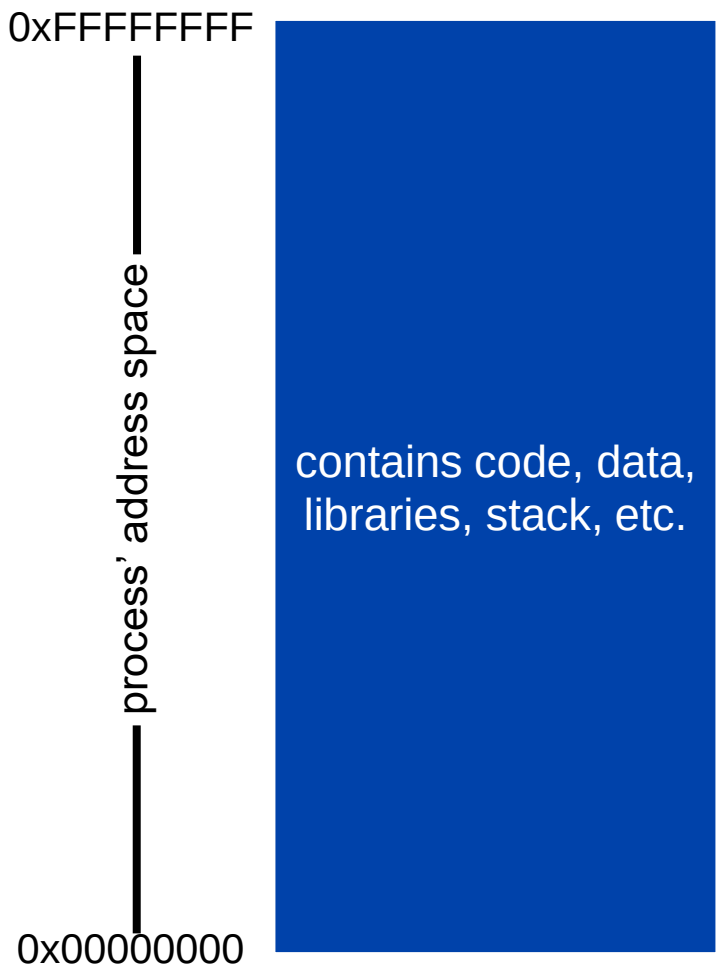


Processes and virtual memory

PennState

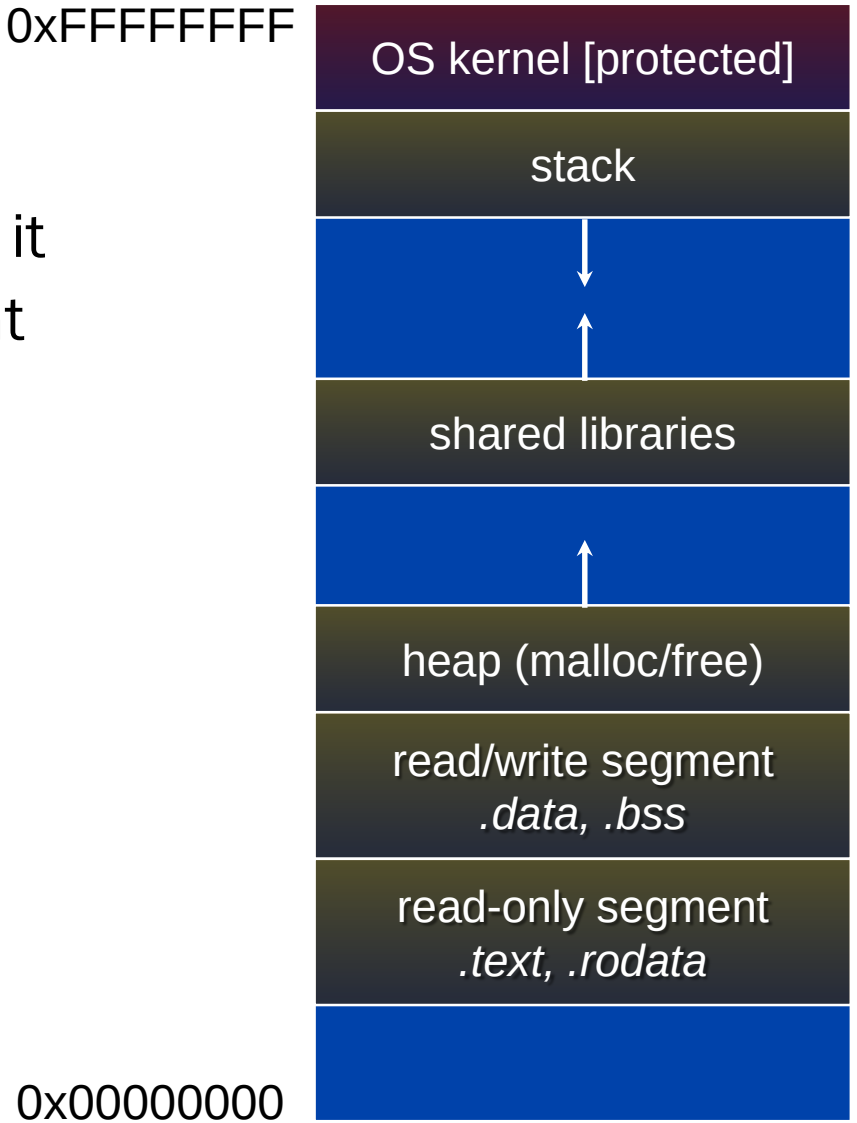
- OS gives each process the illusion of its own, private memory
  - called the process' address space
  - contains the process' virtual memory, visible only to it
  - 32-bit pointers on 32- bit machine
  - 64-bit pointers on 64- bit machine



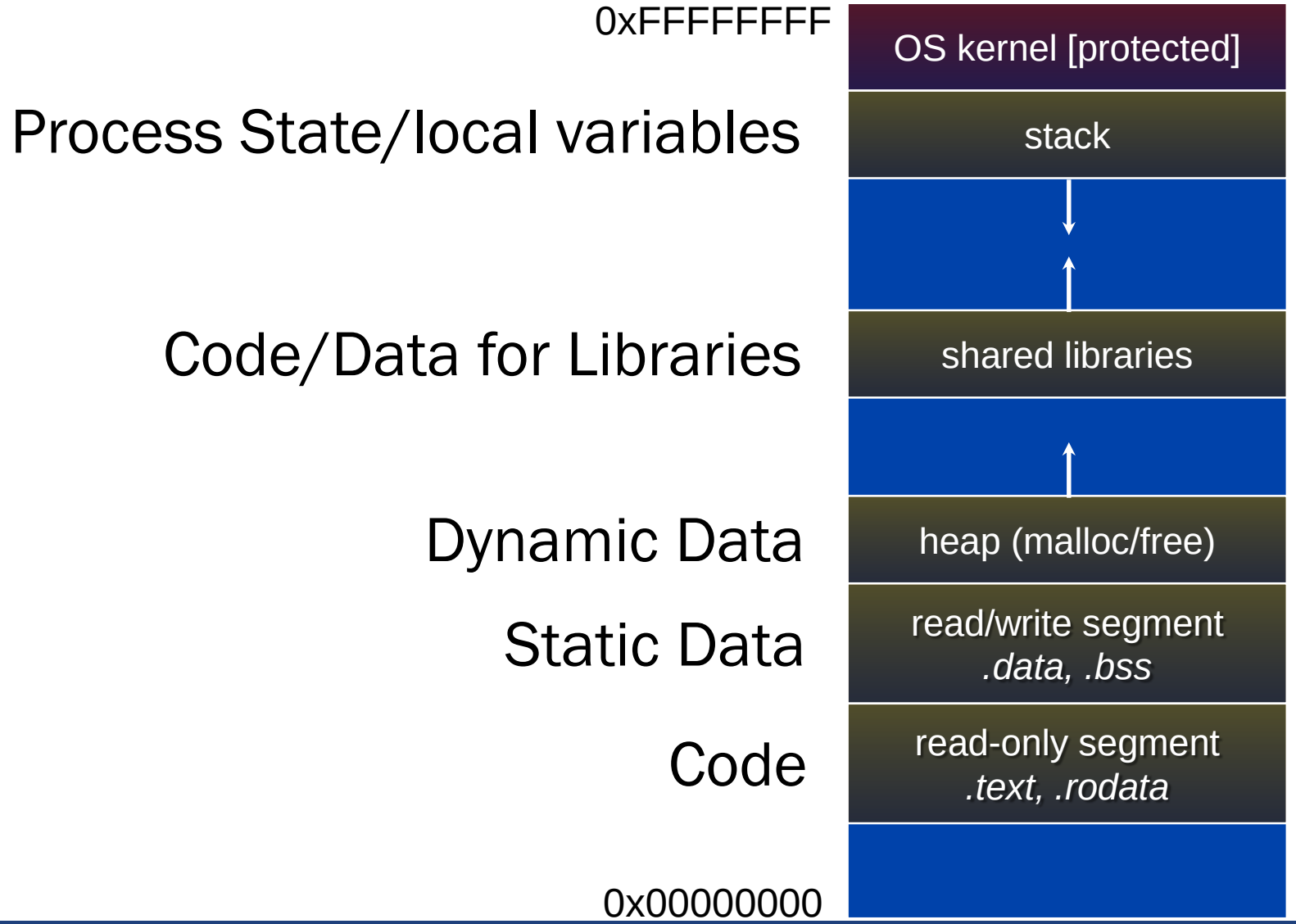


Loading

- When the OS loads a program, it:
  - creates an address space
  - inspects the executable file to see what's in it
  - (lazily) copies regions of the file into the right place in the address space
  - does any final linking, relocation, or other needed preparation



Loading


0xFFFFFFFF
Process State/local variables
Code/Data for Libraries
Dynamic Data
Static Data
Code
0x00000000
OS kernel [protected]
stack
shared libraries
heap (malloc/free)
read/write segment
.data, .bss
read-only segment
.text, .rodata

Something Curious

- Let's try running addr program several times:

```
neo@ubuntu~$ gcc -Wall addr.c -o addr
neo@ubuntu~$ ./addr
x    is at 0x7fff8ed15588
y    is at 0x7fff8ed1558c
a[0] is at 0x7fff8ed15590
a[1] is at 0x7fff8ed15594
foo  is at 0x561785be9169
main is at 0x561785be917c
neo@ubuntu~$ ./addr
x    is at 0x7ffe944b1ee8
y    is at 0x7ffe944b1eec
a[0] is at 0x7ffe944b1ef0
a[1] is at 0x7ffe944b1ef4
foo  is at 0x5575250db169
main is at 0x5575250db17c
```

ASLR

- Linux uses address-space randomization for added security
  - Linux randomizes:
    - executable code location
    - base of stack
    - shared library (mmap) location
  - makes stack-based buffer overflow attacks tougher
  - makes debugging tougher
  - google “disable Linux address space randomization”