CMPSC 297 - Introduction to C Programming

PennState

Systems and Internet Infrastructure Security Laboratory

Week #3

Professor Patrick McDaniel

How Computers Work: Memory

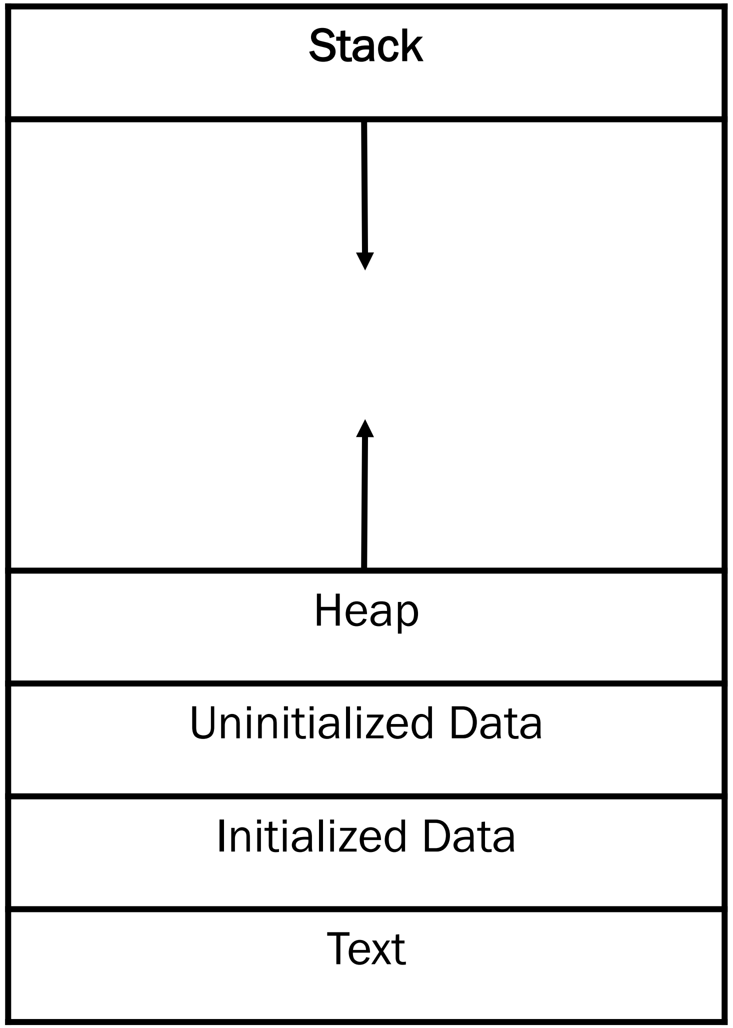
- Memory is a table. Each line has an address and content
  - Everything in memory is tied to an address
  - To access memory, we access the address that holds that information

&x = (address of x)

ADDRESS	VALUE

Anatomy of C Program Memory

- Memory space:
  - **Stack** : contains method and function parameters, return addresses, and local variables **(TODAY)**
  - **Heap** : dynamically allocated data (more on this next week)
  - **Uninitialized Data/Initialized Data** : global and static variables
  - **Text** : instruction codes



Stack

Heap

Uninitialized Data

Initialized Data

Text

Example 1: Memory in a C program

```c
int func1(int x) {
  int y = 2 - x;
  return y + 1;
}

int main() {
  int a = 1, b = 2;
  char c = '!’;
  b = func1(a);
  printf("a = %d, b = %d, c = %c\n", a, b, c);
}
```

NAME	ADDRESS	VALUE
a		
b		
c		
x		
y		

New GDB Commands

`bt full` – shows stack and locals

`up` – view calling function

`down` – undo going up

Old GDB Commands

`p x` – Print the value of x

`p &x` – Print the address of x

`next` – next line

`step` – Step into function

Storing addresses: Pointers

- Recall: Memory maps addresses to values
- What if we want to **store** an address?

- Solution: pointers!
  - Example: `x = &a`

    Translation: “Set `x` to the address of `a`”

    `x` now contains a “reference” to `a`

NAME	ADDRESS	VALUE
a	0x7fffffffdfc8	1
x	0x7fffffffdf88	0x7fffffffdfc8

Storing addresses: Pointers

- Recall: Memory maps addresses to values
- What if we want to **store** an address?

- Solution: pointers!
  - Example: `x = &a`

    Translation: "Set `x` to the address of `a`"

    `x` now contains a "reference" to `a`
  - Dereference: `y = *x`

    Translation: "Dereference `x` and store to `y`"

NAME	ADDRESS	VALUE
a	0x7fffffffdfc8	1
x	0x7fffffffdf88	0x7fffffffdfc8
y	0x7fffffffdf9c	1

Storing addresses: Pointers

- Recall: Memory maps addresses to values
- What if we want to **store** an address?

- Solution: pointers!
  - Example: `x = &a`

    Translation: "Set `x` to the address of `a`"

    `x` now contains a "reference" to `a`

  - Dereference: `y = *x`

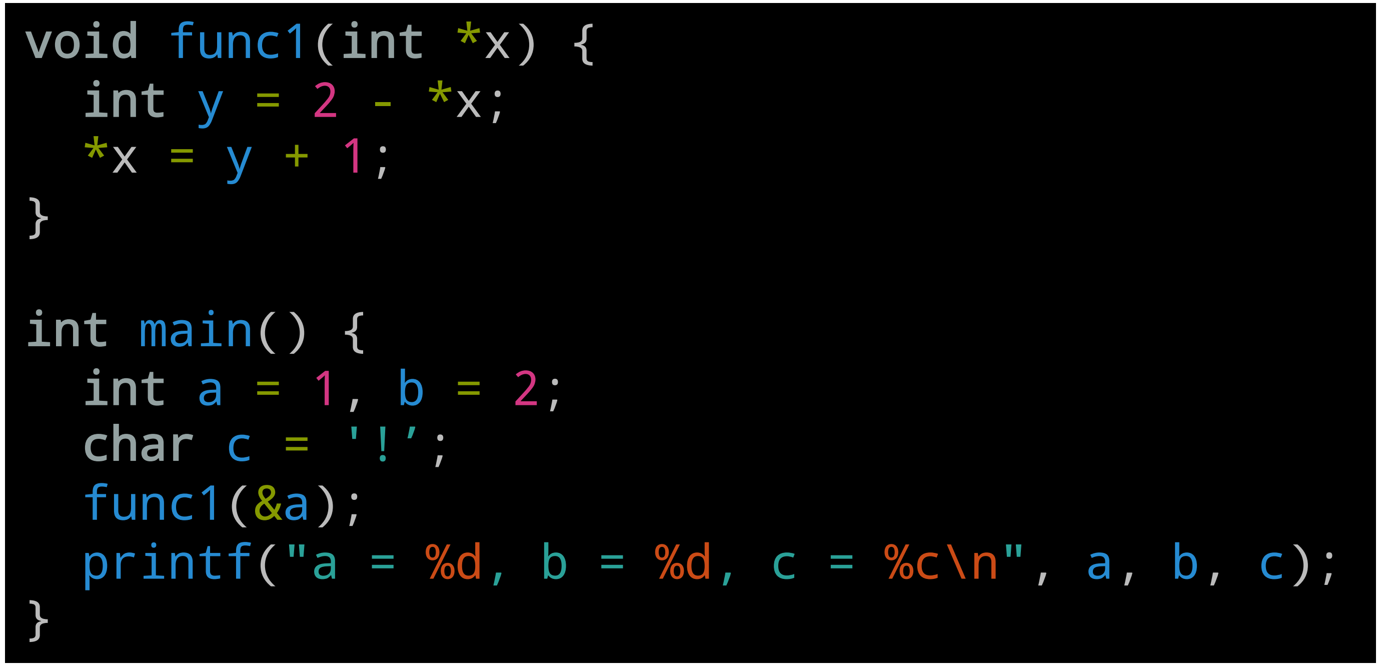
    Translation: "Dereference `x` and store to `y`"
  - Dereference and Set: `*x = 2`

    Translation: "Store 2 in the address pointed to by x"

NAME	ADDRESS	VALUE
a	0x7fffffffdfc8	2
x	0x7fffffffdf88	0x7fffffffdfc8
y	0x7fffffffdf9c	1

Example 2: Pointers

- Follow along to fill out example2-values.txt



```c
void func1(int *x) {
  int y = 2 - *x;
  *x = y + 1;
}

int main() {
  int a = 1, b = 2;
  char c = '!’;
  func1(&a);
  printf("a = %d, b = %d, c = %c\n", a, b, c);
}
```

NAME	ADDRESS	VALUE
a		
b		
c		
x		
y		

Arrays

Arrays

- What if we want to store multiple (n) variables?
  - Just store them next to each other?
  - Call them a[0], a[1], a[2], … a[n-1]
- Each has its own address (just like before)
  - **But**, we can refer to them using numbers

NAME		ADDRESS	VALUE
a	a[0]	0x7fffffffdfb0	
	a[1]	0x7fffffffdfb4	
	a[2]	0x7fffffffdfb8	
	a[3]	0x7fffffffdfbc	

- Example: `int a[4]`
  - Translation: An array of 4 integers called `a`

- Getting addresses: `&a[0]`
- Array variables refer to the address of their 0th element (e.g., `a == &a[0]`)

Accessing Array Members/Addresses

- Example: `int a[4]`

- Getting elements: `a[0]`
- Setting Elements: `a[0] = 1`

- Address of elements: `&a[0]`

- Array variables are pointers to the address of the 0th element: `a == &a[0]`

NAME		ADDRESS	VALUE
a	a[0]	0x7fffffffdfb0	
	a[1]	0x7fffffffdfb4	
	a[2]	0x7fffffffdfb8	
	a[3]	0x7fffffffdfbc	

Passing Arrays

- Recall: array variables are just addresses
  - So we can pass them just like we pass pointers
  - And we can treat pointers as arrays!

NAME		ADDRESS	VALUE
a, x	a[0], x[0]	0x7fffffffdfb0	
	a[1], x[1]	0x7fffffffdfb4	
	a[2], x[2]	0x7fffffffdfb8	
	a[3], x[3]	0x7fffffffdfbc	

- Example:

```
void func(int *x)

int a[4]
func(x)
```

- In `func`: `x` is **the same address** as a

Example 3: Arrays

- Follow along to fill out example3-values.txt

```
void func1(int *x, int size) {
  int i;
  for (i = 0; i < size; i++) {
    x[i] = x[i] + 2;
  }
}

int main() {
  int a[] = {1, 2, 3, 4};
  func1(a, 4);
  printf("a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
    a[0], a[1], a[2], a[3]);
}
```

NAME		ADDRESS	VALUE
a	a[0]		
	a[1]		
	a[2]		
	a[3]		
x			
size			
i			

Strings

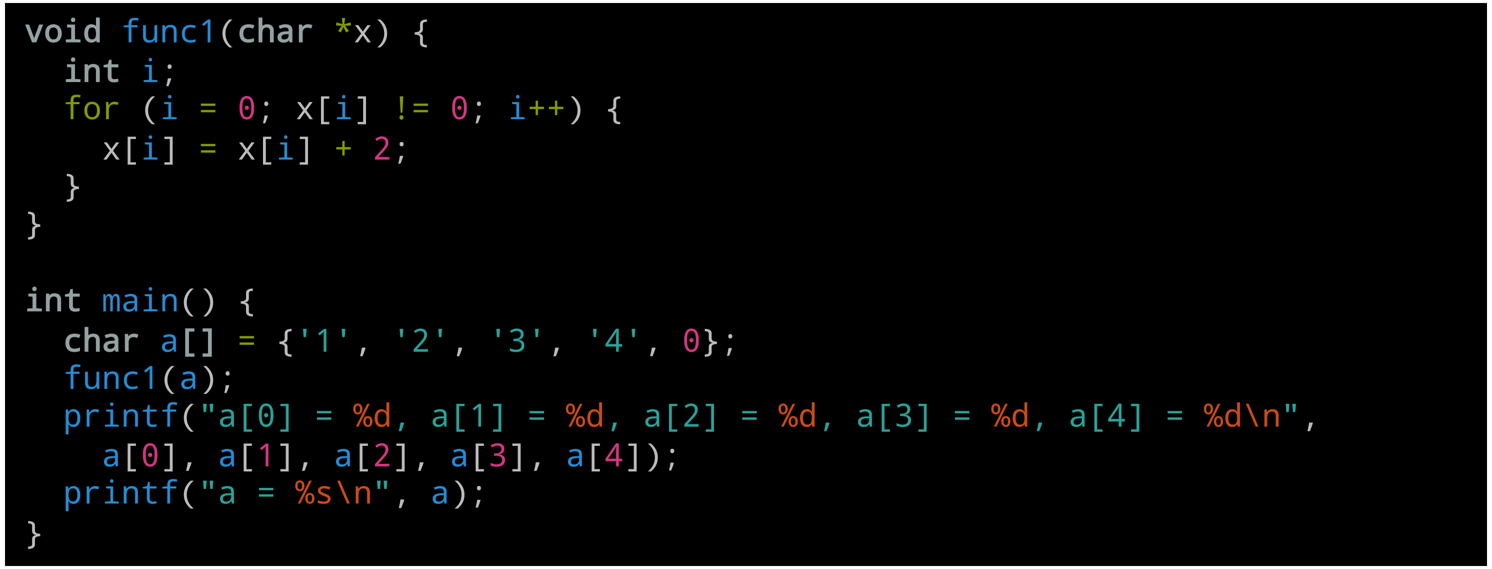
Strings

- Before: passed size of array to function
  - What if we don’t want to pass the size?
- Trick: put some “special” value at the end of the array
  - So function knows when to stop

- Especially useful for character arrays (strings): end array with 0

Exercise: Strings

- Use GDB to fill out example4-values.txt



```
void func1(char *x) {
  int i;
  for (i = 0; x[i] != 0; i++) {
    x[i] = x[i] + 2;
  }
}

int main() {
  char a[] = {'1', '2', '3', '4', 0};
  func1(a);
  printf("a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d, a[4] = %d\n",
    a[0], a[1], a[2], a[3], a[4]);
  printf("a = %s\n", a);
}
```

NAME		ADDRESS	VALUE
a	a[0]		
	a[1]		
	a[2]		
	a[3]		
	a[4]		
x			
i			