CS 240: Programming in C

Lecture 11: Introduction to Pointers

PURDUE UNIVERSITY

Announcements

- Guest lecture on Monday
    - It’s important -- don’t miss it!
- Wednesday lecture next week is cancelled
- My office hours next week are cancelled
    - TAs will still hold office hours

Announcements

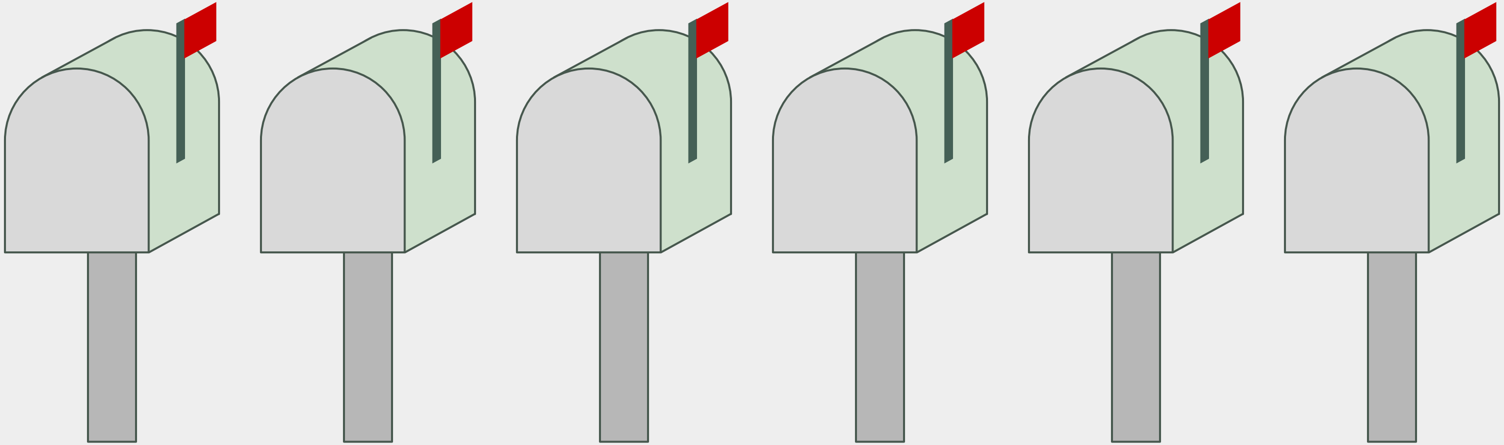
- Homework 4 due tonight!

Learning about pointers

- Read Chapter 5 of K&R. ALL OF IT
  - …and/or Beej Chapter 7
- Understanding basic pointer concepts is easy
- Understanding the combination of concepts is harder
- Applying the concepts in a meaningful manner takes patience and practice
- We’ll look at one basic concept at a time

Memory

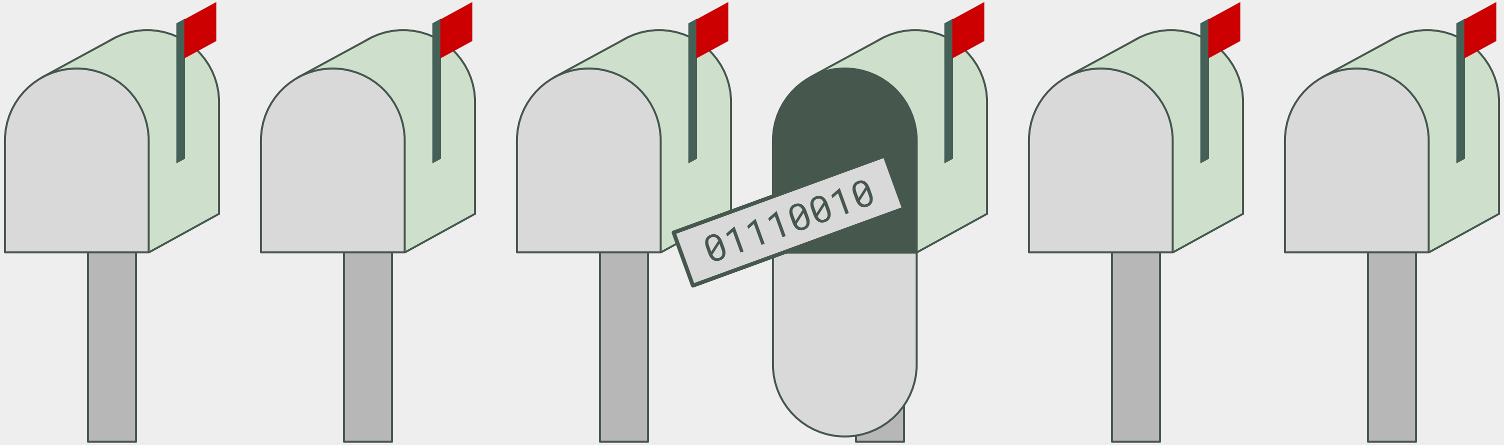
- Computer memory is organized in a contiguous straight line, like a row of mailboxes on a **very** long road



Memory values

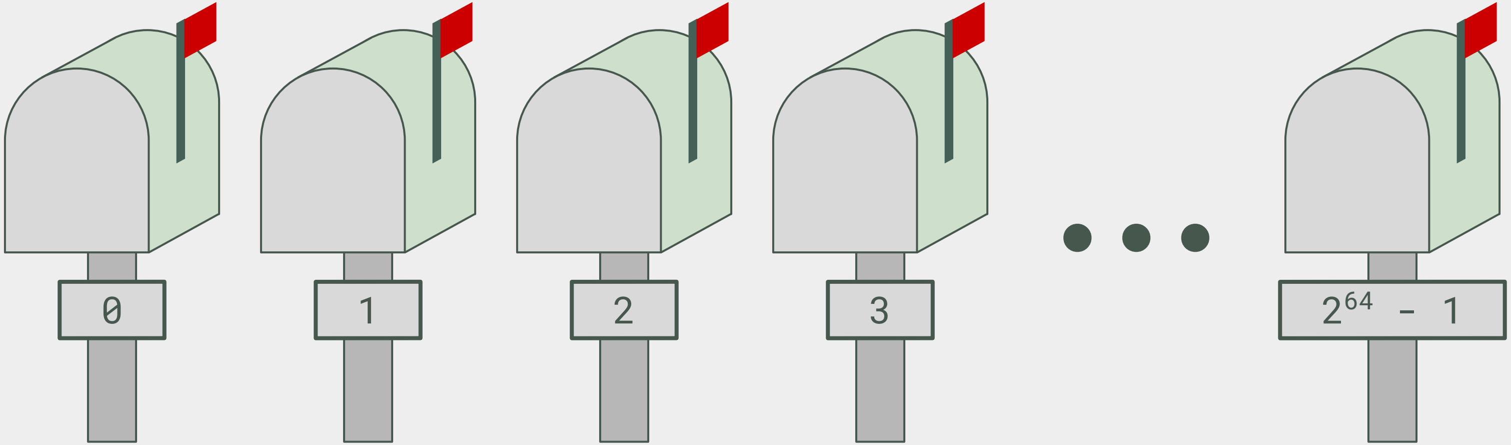
- Each mailbox can hold one byte (8 bits)
- Can represent a value between 0 and 255
- Or a single character

```
01110010 = 0x72 = 114 = ‘r’
```



Memory addresses

- Every mailbox has an address. To put something in a mailbox, you need to know its address. Our addresses go from 0 to 18,446,744,073,709,551,615 (2^{64} - 1)



What is a pointer?

- Simply, a pointer is a **variable** that stores the **address** of another variable

Normal variables

- When we say…

```
int x;
```

the x variable occupies some space in memory.

- When we have a statement like…

```
x = 5;
```

the compiler “knows” how to compute the address of the x variable and can arrange to have a value put there

How to get the address?

- The C language has an operator **&** used to determine the location of **any storage element**.

  For example:

  ```
  p = &x;
  ```

  p now holds the address that x resides at

- p is not equivalent to x!
  - Instead, we say that **p points to x**

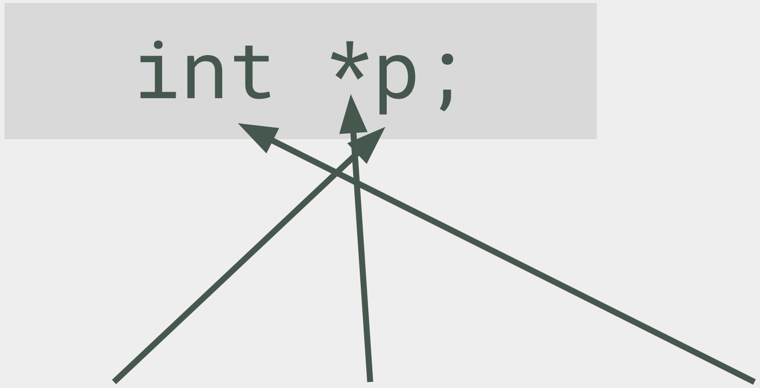
Address-of

```
#include <stdio.h>

int main() {
  int x = 0;

  printf(“Address of x: %p\n”, &x);

  return 0;
}
```

```
Address of x: 0x7ffdfcda5404
```

How do we define a pointer?

- The creation of a pointer looks like this:



```
int *p;
```

This means p is a pointer to an integer

How to manipulate a pointer?

- Now that we can find out the address of a variable, how can we use that address?
- The C language has an operator * called the dereference operator, or **contents-of** operator.

  For example:

  ```
  *p = 5;
  ```

  will place the value 5 into the memory location pointed to by p

Definition vs dereference

- The * has multiple meanings, depending on context

```
int x = 7;
int *p = &x;    /* This * means p is a pointer */
```

```
*p = 5;         /* This * means dereference p */
```

```
int y = x * 4;  /* This * means multiply x */
```

A complete example

```
#include <stdio.h>

int main() {
  int x = 0;
  int *p = 0;

  p = &x;
  *p = 5;

  printf(“x = %d\n”, x);
  return 0;
}
```

p now “points” to x

*p is equivalent to x

Why do we need pointers?

- Something we **cannot** do with variables:

```
#include <stdio.h>
void increment(int n) {
  n = n + 1;
}

int main() {
  int x = 0;
  increment(x);
  printf(“The value of x = %d\n”, x);
  return 0;
}
```

```
The value of x = 0
```

It didn't work

- Why wasn't x incremented?
  - x was **not** passed to the increment function
  - The **value of x** was passed to the increment function
  - The new value computed inside increment was not stored back into x
- In fact, there is no way for increment() to modify the value of x
  - We call this "pass-by-value"
- We need some way to tell increment() about the memory location of x

What is a pointer good for?

- Let’s use a pointer instead…

```
#include <stdio.h>
void increment(int *p) {
  *p = *p + 1;
}

int main() {
  int x = 0;
  increment(&x);
  printf(“The value of x = %d\n”, x);
  return 0;
}
```

```
The value of x = 1
```

Now you understand scanf()

- Now you know why you have to use the **&** operator when you use scanf(), e.g.,

```
scanf(“%d\n”, &x);
```

  - You’re not passing the variable to scanf
  - You’re telling scanf what the address of the variable is, so that scanf can fill it in
- This is called passing by reference, passing by pointer, or passing by address
- Why don’t we need **&** when we pass an array?

Pointers can be used as arrays

- When you obtain the address of a variable…

```
ptr = &x;
```

- You can “dereference” it two ways:

```
y = *ptr;    /* treat as pointer */
z = ptr[0];  /* or as 1 element array */
```

- The effect and meaning are exactly the same

Using a pointer as an array

```
#include <stdio.h>

void inc(int *ptr) {
  ptr[0]++;    /* use as array */
}

int main() {
  int num = 0;
  inc(&num);   /* pass as a pointer */
  printf(“num = %d\n”, num);
  return 0;
}
```

Arrays are equivalent to pointers

- When you assign an array (not one of its elements) to something, you're assigning a pointer:

```
ptr = array;
```

- When you pass an array (not one of its elements) to something, you're passing a pointer:

```
strcpy(array1, array2);
```

- When you return an array, you return a pointer:

```
return array;
```

Example

```c
#include <stdio.h>

int *zap(int *ptr) {
  ptr[0] = 0;
  return ptr;
}

int main() {
  int array[100];
  int *ptr = 0;
  ptr = zap(array);
}
```

Arrays vs. pointers

- You can assign something new to a pointer, but an array always points to the same thing

```
ptr = array;   /* OK */
array = ptr;   /* Not allowed! */
```

- An array definition allocates space for all the elements, but not the “pointer”
- A pointer definition allocates space only for the pointer value (address, 8 bytes)
- A function parameter defined as an array is really just a pointer

Array function parameter

```c
int sum(int array[2]) {
  int s = 0;

  for (int i = 0; i < 50; i++) {
    s = s + array[i];  /* is this legal? */
  }
  return s;
}
```

Address-of array elements

- Since an array is already an address, it makes no sense to find the address of an array

```
ptr = &array;
```

- But you can find the address of an element

```
ptr = &array[3];
```

Address-of array elements

- You could also say:

```
ptr = array;    /* Address of array[0] */
ptr = ptr + 3;  /* go to 3rd element */
```

- Or even:

```
ptr = &array[0];
ptr++;
ptr++;
ptr++;
```

This is called pointer arithmetic

- You can add or subtract constants

```
ptr = ptr + 1;    ptr = ptr - 12;
```

- You can increment / decrement

```
ptr++;  ptr--;  ++ptr;  --ptr;
```

- You can even subtract one pointer from another

```
int arr[100], *ptr1, *ptr2;
long diff;
ptr1 = &arr[10];
ptr2 = &arr[20];
diff = ptr1 - ptr2;
```

Another pointer example

```
int main() {
  int ctr = 0;
  int *ptr = 0;
  int int4 = 18;
  int int3 = 11;
  int int2 = 10;
  int int1 = 7;

  ptr = &int1;

  for (ctr = 0; ctr < 7; ctr++) {
    printf("Value at address %p: 0x%x (%d)\n",
           ptr, *ptr, *ptr);
    ptr++;
  }

  return 0;
}
```

For next lecture

- Read K&R Chapter 5, Beej Chapter 7
- Study the examples in this lecture at home
- Practice the examples
- Modify the examples

Slides

- Slides are heavily based on Prof. Turkstra’s material from previous semesters.