C pointers

- What are C pointers?

  - Answer: *variables* that store memory *addresses*

    - i.e., they “point” to memory locations

    - And they can vary – be assigned a new value
int m = 37;

The C compiler has to make a lot of things happen to deal with this simple statement:

• A memory location is allocated for the variable m
• During the execution of the program, an instruction that stores the value 37 to that memory location is executed
• Later on in the program if we see the identifier m (in the same scope), then we know that we are referring to either
  – to the value stored at the memory location allocated for m, or
  – to the memory location allocated for m
The address operator: &

- Can we access the lvalue (i.e., the memory address) of a variable on the right-hand-side of an assignment (or in an expression in general)?
  - Yes, we can do that using the **address operator &**
  - Given a variable \( x \),
  - \&x \ is the lvalue of the variable \( x \)
  - Equivalently, \&x \ is the memory address of \( x \)

We have seen the \& \ operator before:

```c
scanf("%d", \&x);
```

- This function call is telling the `scanf` function to store the value it reads from the `stdin` input stream to the memory location \&x
The dereference operator: *

- C language allows us to declare variables that store memory addresses: They are called pointers.
- Given a memory address, the dereference operator * returns the value stored at that memory address.

- We use * to declare pointer variables:
  
  ```c
  int *p; /* p points to an int, i.e.,
  dereferencing p returns an int value */
  ```
Using * and & operators

• Use * to declare a pointer variable:
  ```c
  int *p; /* now p can point to an int */
  ```

• Use & to retrieve a variable’s lvalue:
  ```c
  p = &m; /* p is now storing the address of m i.e., p is pointing to m */
  scanf("%d", p); /* stores input in m */
  ```

• Use * to dereference a pointer:
  ```c
  *p = 19; /* stores 19 at the location p points to */
  printf("an int value: %d", *p);
  /* prints the value that p is pointing, i.e., prints the value stored at the memory location p */
  ```
Pointer types

- For dereferencing, and for pointer arithmetic: The type that a pointer points to must be known
  - e.g., an `int *` can only point to an `int`
  - Exception: a `void *` can point to any type
    - So cannot dereference directly – must cast first

- Can a pointer point to a pointer?
  - Why not? Pointers are variables too!
  ```c
  double d, *dp = &d, **dpp = &dp;
  **dpp = 17.5;  /* stores 17.5 in d */
  ```
Pointer (address) arithmetic

• The operations that can be performed on pointers (or addresses):
  – A pointer can be assigned to another pointer of the same type
  – An integer value can be added to or subtracted from a pointer
  – A pointer can be assigned to or compared with the symbolic constant NULL which is defined in <stdio.h>
  – Pointers to elements of the same array can be subtracted or compared as a means to accessing elements in the array
• Cannot use multiplication or division with pointers
• Cannot add two pointers either
Incrementing a pointer

- Incrementing a pointer does not necessarily increase the address value by one
  - It increases it based on the size of the memory required to store the value it points to.

Execute the following code and see what happens:

```c
char *cptr=(char *)1000;
int *iptr=(int *)1000;
double *dptr=(double *)1000;

cptr++;    // Incrementing a pointer may not increase the address value by one
iptr++;    // It increases it based on the size of the memory required.
dptr++;    // Here, dptr is incremented by 8 times its original address.

printf("New value of char ptr : %u\n",cptr);
printf("New value of int ptr : %u\n",iptr);
printf("New value of double ptr : %u\n",dptr);
```
Array names are *not* pointers, but they are like pointers

```c
int x[10];
```

What does this declaration do?

- Allocates 10 *consecutive* `int` locations
- Also *permanently* associates `x` with the address of the first of these `int` locations – i.e., `x` is the same as `&x[0]`

- `&x[i]` is exactly the same as `(x+i)`
- `x[i]` is exactly the same as `*(x+i)`

- So why is `x` not a pointer?
  - You cannot modify the value of `x`, it is always pointing to the first element of the array
Example

- The two loops below are doing the same thing

```c
int x[] = { 1, 2, 3, 4, 5, 6};
int *ptr = x;
int sum1 = 0, sum2 = 0;
for (k = 0; k < 6; k++) {
    sum1 += x[k];
}
for (k = 0; k < 6; k++) {
    sum2 += *(ptr + k);
}
printf("Sum is %d and %d\n", sum1, sum2);
```
More pointers vs. array names

- If p is int* (i.e., a pointer to int), then …
  - p = &x[0] is okay
  - And p = x has exactly the same effect

- Moreover p[i] is now an alias for x[i]
  - Until ++p moves p to point at x[1] or beyond

- But x = p is illegal (assuming x is an array name)
Parameter passing with pointers

- In C variables always passed to functions “by value”
  - So functions need pointers to change values
    ```c
    change(x);
    /* function cannot change x’s value */
    change(&x);
    /* function may change x’s value */
    ```

- Return values are copies too – so similar issues
void triple1(int x) { x = x * 3; }
void triple2(int *x) { *x = *x * 3; }

• Later in main function, for instance:
  int a[] = {10, 7};
  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
Why do we have to deal with pointers?

Given that programming using pointers seems somewhat challenging, why do we need them?

One reason is, as we have seen earlier, pointers allow us to write functions with side-effects, i.e., the changes made in the function body are reflected to the caller.

You have to use this carefully since side-effects are hard to understand when someone else looks at your code.

A more significant reason for using pointers is that, without pointers, we would be forced to know the size of each memory we plan to allocate before runtime.

We would need to know the sizes of the arrays we plan to allocate for example.

But sometimes, the size of an array we plan to allocate may depend on the size of the input. What can we do?
Two ways to allocate memory

**Static memory allocation** – done at compile-time

```c
int x; double a[5];
/* allocates space for 1 int, 5 doubles */
```

Both size and type are clearly specified ahead of time – x can only hold int values, a only doubles

**Dynamic memory allocation** – done during execution

Must use library methods like malloc

- `malloc` allocates specific amount of memory, returns `void *`

Cast to appropriate pointer type – then use as always

```c
int *ip = (int *)malloc(sizeof(int));
```

- Notes: returns NULL if memory not available so must check that; cast is optional

Must **free** the memory when done with it: `free(ip);`
What is `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

• Works for arrays too – total bytes in whole array
The `malloc` function

- `malloc` is used for dynamic memory allocation.
  The argument of the `malloc` function is the number of bytes of memory required (where each byte is 8 bits).
  For example, if we want to allocate a memory for 10 integers, we need to figure out how many bytes we need per integer and then multiple that by 10.

  We can use `sizeof(int)` to get the number of bytes used per integer value.
  - Note that this value is system dependent

- If there is not enough memory for allocation, then `malloc` returns NULL.
- `calloc` : like `malloc` but initializes the allocated memory to 0.
- `realloc` : can be used to change the size of the allocated memory.
Pointer tutorial

Pointer fun

http://www.cs.ucsb.edu/~mikec/cs16/slides/binky.html
Strings in C

- A char array, terminated by ' \0 ' 
- String literal – anything enclosed in double quotes  
  – e.g., "cat" – automatically includes terminator

| c | a | t | \0 |

- Must allocate memory to store the characters
  char sentence[50];
  /* up to 49 chars plus ' \0 ' */
char word[] = "cat";
  /* length is 3, but size is 4 */
- Point to such memory with char* (pointer to char)
  char *s = sentence;
s = word;
Strings in C

Equivalent declarations of a string

```c
char filename[] = "sensor.txt";
char filename[] = {'s', 'e', 'n', 's', 'o', 'r', '.', 't', 'x', 't', '\0'};
```
String library functions

The standard C library contains a number of functions for strings which are defined in `string.h`:

- `strlen(t)`: Returns the length of the string `s`.
- `strncpy(s, t)`: Copies string `t` to string `s` and returns a pointer to `s`.
- `strcat(s, t)`: Concatenates string `t` to the end of string `s`.
- `strcmp(s, t)`: Compares string `s` to string `t` in an element-by-element comparison. A negative value is returned if `s < t`, zero is returned if `s` is equal to `t` and a positive value is returned if `s > t`;
Topics we have discussed after the first midterm

File IO
Functions
Parameter passing
Scope, global vs. local variables
Recursive functions
Arrays
Multi-dimensional arrays
Arrays as function parameters
Pointers
Address and dereference operators
Pointers and arrays
C strings
dynamic memory allocation