0. Introduction

Last week we looked at two types of variables: strings and structs. We saw how to model a train trip by using an array of structs, each struct storing information about the name, time, direction of the train stopping. This is one way of organizing memory.

We saw that a string is just a region of memory that contains a sequence of characters terminated with an element with value zero. We saw how overrunning the bounds of the array would deposit data in memory locations assigned to other variables or other sorts of information.

This week we look at the details of variables and where they are stored in memory. C lets us refer to regions of memory in two ways: through variable names, and through memory addresses.

The outline for tonight’s class is:

1. Quick review of the big picture
2. Data types and structures
3. A problem: finding palindromes
4. Computer Memory: storage, addresses, pointer variables and their operations
5. Pointers and strings
6. Pointers to structs
7. Pointers to arrays
8. Pointers and functions
9. Practice
10. Arrays and Functions
11. Pointers and Strings: strlen and strcpy
12. Brackets and Stars: Indexing vs Pointing
13. Arrays of Pointers
14. Pointers to Functions
15. Summary

1. Quick Review

The big picture for C/Unix/CGI programs is:

2. Data Types and Data Structures in C

First, a quick look back at the types of values we can store and the structures we can build. The basic types match machine types exactly:

- `int` -- short, long, unsigned, long long
- `char` -- unsigned
- `float` -- double

C offers only two ways to combine values:

- `array` -- a sequence of contiguous variables, all one type, indexed by offset
- `struct` -- a combination of any types, indexed by a tag

C offers one more data type: the pointer. Today’s class is about pointers. We shall look at the idea, how to
define them, what operations they support, and how to use them with the other types and structures.

3. Introduction: Finding Palindromes

The file `/usr/share/dict/words` contains nearly 100,000 words. Which of these words are palindromes? What is a palindrome? How could you find them?

The logic is pretty simple:

```c
read in a line
if no more data, break
if line is a palindrome
print it
loop back
```

To read in a line, we need a place in memory to hold it. We use an array of chars. To read in the line, we use a function, `fgets()`. What about testing it? We need to write a function. As mentioned before, arrays are always passed by reference, so even long words are passed without time- or space-consuming copying.

The logic of `is_a_pal` is pretty simple. We put two fingers on the blackboard, one at the first char, one at the last one. We compare the char each finger points to. If they match, we move the fingers close to each other and repeat until we get a mismatch or until the fingers meet or cross.

We can of course use array notation and indexing, but at the lower level, we are really just pointing to locations in memory. In C, the notation `a[i]` just means "the value stored in the element `i` places past the array that starts at location `a`."

C allows us to ask where a variable is and to work with those addresses as actual data values. Many higher level languages protect you/prevent you from being able to work at this level.

Therefore, to make use of this feature of C, we need to learn:

1. How computer memory works
2. How to find the address of a variable
3. How to use the address of a variable

4. How Computer Memory Works: An Array of Little Boxes

Computer memory is a linear sequence of storage cells. Each storage cell (called a memory location) holds a single 8-bit quantity (a value between 0 and 255). The cells are numbered from 0 to the amount of memory in the computer. Every variable you define is assigned a block of locations, large enough to hold the type of value you want to store there. When the computer executes a statement like `x = 2;`, the computer stores the value 2 in the memory locations assigned to the variable `x`.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 ...
```

For example, the code:

```c
int x;
```

tells the compiler you want an integer-sized storage region and you want to call it `x`. On 32-bit machines, this variable usually will be assigned 4 storage locations, and on a 64-bit machine, this variable will be assigned 8 storage locations. The first location in the block is the `address of the variable`.

```
x
3068 ...
```
4.1. Finding the Address of a Variable: &X

C allows you to obtain the actual memory address of any variable. The operator & obtains the address of a variable. For example [show-addrs.c]:

```c
#include <stdio.h>
/*
 * show how the & operator can be applied to any varname
 */
struct person {
  int age;
  char name[30];
  char city[30];
};
int main()
{
  int x;
  char c;
  float a;
  char l[10];
  struct person p = { 12, "Lee", "Boston"};
  /* simple vars */
  printf("x is at %u, c is at %u, a is at %u\n", &x, &c, &a);
  /* arrays */
  printf("l is at %u, l[2] is at %u, l[100] is at %u\n", &l, &l[2], &l[100]);
  /* structs */
  printf("p is at %u, p.age is at %u, p.name is at %u, p.city is at %u\n", &p, &p.age, &p.name, &p.city);
}
```

produces, on one machine, the results:

```
x is at 1180043044, c is at 1180043055, a is at 1180043048
l is at 1180043120, l[2] is at 1180043122, l[100] is at 1180043220
p is at 1180043056, p.age is at 1180043056, p.name is at 1180043060, p.city is at 1180043090
```

Those are the actual memory locations assigned to those variables. Say you wanted to record those addresses? What kind of variable would you store them in? You could use an int, but there is a type of variable in C designed specially to hold addresses of variables: a pointer variable.

4.2. Pointer Variables: Storing Addresses of Things

A pointer variable is a variable that stores the address of a section of memory. We have seen int variables, they store integer values. We now consider pointers to int variables. A pointer to an int stores the address of an int variable. A char variable holds a character, and a pointer to a char variable holds the address of a char variable.

4.3. Pointers: Defining Pointers

Here is the syntax for defining pointer variables:

```c
int *p1;       /* create a pointer to an int */
char *p2;      /* create a pointer to a char */
float *p3;     /* create a pointer to a float */
struct tstop *p4; /* create a pointer to a struct */
```

The notation is simple. One puts the type of the pointee, a star, and the name of the pointer variable. For example, the first definition creates a pointer variable that can store the address of an int variable.

4.4. Pointers: Assigning Values to Pointers

Use & to get the address of a variable, and use = to store that address in a pointer variable [assign-ptrs.c]:

```c
int x;        /* an int */
```
int *p; /* a pointer to an int */
int *q; /* another ptr to an int */
p = &x; /* store address of x in p */
x = 12; /* store a value in x */
q = p; /* copy a pointer value */

Using the address of x shown in section 2.1, we draw the diagram as:

```
      p
   1180043044
      q
   1180043044
       x
   12
```

Variable x, an int variable, is stored at location 1180043044 and contains the value 12. Pointer variable p, a pointer to an int, contains the value 1180043044. We say that "p points to x". Another pointer, q, also contains the address of x. We say "q points to x". Note there is only one variable x, but two pointers to x.

4.5. Pointers: Using * to Dereference Pointers

We now have two ways to refer to the value stored in x. First, we may refer to a value by name:

```
x = 100;
x += 4;
printf("x is %d", x);
```

Second, we can refer to the value stored in x by using the pointer:

```
*p = 100;
*q += 4;
printf("x is %d", *p);
```

To summarize, x is the name of the variable, p currently points to the variable x, and *p is the variable pointed to by p. One can say "x is the pointee of p", but that phrase is not a common phrase.

The common term is "dereference". One says the * operator used in *p "dereferences the pointer", yielding the variable pointed to by p.

4.6. Pointers: Other Operations

We now consider the operations that can be used with pointer variables. Before we consider them, remember that a pointer variable is a variable that holds the address of a variable. It holds a number that tells where in memory some value is stored. Its value is an address. Got that?

* Dereference a pointer. Returns the variable pointed to.
= Copy an address from one pointer to another. Afterward, both pointers point to the same place.
== Compare addresses. If both pointers point to the same address, this operation returns true.
!= See ==
<, > Compares addresses. If one address is lower than or greater than another the comparison returns true. Not as useful as == and !=, but meaningful and handy. Sometimes.
+, -, ++, -- See discussion of pointers and arrays below.

We now leave abstractions and look at pointers in practice.

5. Pointers and Strings

We can now write is_a_pal using pointer variables. The logic is as follows:
set one pointer to the first char  
find the last char  
set other pointer to point to last char  
while fingers do not cross  
if values do not match, return false  
moves pointers

Here is the code  
/* returns 1 if arg is a palindrome, 0 if not */  
/* note: "" IS a palindrome */  
/* TODO: make it case insensitive */

```c
int is_a_pal(char s[])  
{  
    char *left, *right;  
    left = &s[0];  
    /* find end of string */  
    for(right=&s[0]; *right != \n' & *right != \0 ; right++ )  
    {  
        right--;  
    }  
    /* do the check for palindrome */  
    while( left < right )  
    {  
        if ( *left != *right )  
            return 0;  
        left++;  
        right--;  
    }  
    return 1;  
}
```

6. Pointers to Structs

Consider this code:

```c
struct person {  
    int age;  
    char name[30];  
    char city[30];  
};  

struct person *pp; /* pointer to person */  
(*p).age = 22; /* set age member to 22 */  
p->age = 22; /* nicer notation */
```

You can create pointers to structs and use those pointers to refer to the structs and to the members of the structs.

7. Pointers to Arrays

Consider this code:

```c
int a[5] = { 2, 4, 1, 3, 9 }; /* make an array */  
int *ap; /* make a pointer */  
ap = &a[0]; /* point to array */  
*(ap + 2) = 9; /* pt to element */  
ap[2] = 9; /* nicer notation */
```

8. Pointers and Functions

Pointers are values, just like ints, floats, and chars. We can pass them to functions, and functions can return them.
9. Practice

Now for some exercises. Work through the code in ptr1, ptr2, ptr3, ptr4, and ptr5 samples to see how well you understand how this notation works.

10. Arrays and Functions

In previous classes, we said that arrays are always passed by reference. This is no longer mysterious.

The simple fact is:

**The name of an array is its starting address.**

The name of an array is just an address, a number, like 4 or 23027. The name of an array is not a variable. It is just an address. As such it can be stored in a pointer variable.

Thus:

```c
int a[10];
int *p;
p = a; /* store address of a in p */
```

is legal and commonly used.

Thus, when you pass an array to a function, you are just passing an address. In the function, that address is stored in the formal parameter, which is therefore a pointer variable. The notation:

```c
int f(char a[])
{
    ...
}
```

Says the value being passed in is an array, but it really means that a is a pointer variable. You just as easily write that code as:

```c
int f(char *a)
{
    ...
}
```

which is what the compiler does.

11. Pointers and Strings: strlen and strcpy

Pointers are frequently used when working with arrays, and, in particular, with strings. Consider `strlen`, the standard library function. Here is one version of strlen: `[string_samples.c]`

```c
unsigned strlen_i(char a[])
{
    unsigned pos = 0;
    while( a[pos] != '\0' )
        pos++;
    return pos;
}
```

This example refers to each element by its base and offset. The base is the name of the array (a), and the offset is the index(pos). Compare that to version two of the function:

```c
unsigned strlen_p(char a[])
{
    char *p ; /* define a pointer */
    p = &a[0]; /* point to item 0 */
    while( *p != '\0' ) /* point to null? */
        p++; /* no: increment */
    return p - a; /* compute distance */
}
```

In this pointer-based example, we do not use base and offset. Instead we use a pointer variable. We first store the address of the first element, and dereference that pointer to check the character. We keep
incrementing the address until we point to the end of the string. The difference in addresses from the start to where we find the nul is the length of the string. Use this diagram to trace the code in the second version.

```
array    char a[8];
                ^
   0 1 2 3 4 5 6 7

pointer   char *p
```

Here is a second string function, strcpy using pointers:

```c
char *strcpy_p(char dest[], char src[])
{
    char *dp = dest, *sp = src;
    while( *dp = *sp ){
        dp++;
        sp++;
    }
    return dest;
}
```

This function is C at its almost most terse. You need to draw a picture to trace this function. Note we use the names of the strings as addresses. When an array is passed into a function, the address of the first element is passed to the function. Think about carpets. For several weeks, we have said that strings are arrays of characters terminated by a nul char. What is the connection between arrays and pointers, anyway?

### 12. Brackets and Stars: Indexing vs Pointing

Understanding the differences and connections between arrays and pointers is, for many, the trickiest part of working in C. Refer to this list of facts often until you get the hang of it. The ideas are actually pretty simple. The notation, and its flexibility, is the problem.

**CENTRAL FACT**

\[ A[i] \text{ MEANS } *(A+i) \text{ -- that is, the item at offset } i \text{ from address } A \]

- \( A \) represents an address: an array name or pointer plus/minus any distance

**SUPPORTING FACTS**

1. \( \text{int a[10]} \) allocates a chunk of memory large enough to hold 10 ints
   - [shirts and carpets: creates a 10-room house]
2. The name of an array is a number, a constant -- the address where the chunk starts
3. \( \text{int *p allocates a piece of memory large enough to hold one address} \)
   - [shirts and carpets: creates a place to record the address of one house]
4. \( p = a \) stores in a pointer variable the address of the start of a
5. \( A[i] \) indicates the \( i^{th} \) element from the address \( A \)
   - \( A \) may be the name of an array OR any address value <-- possibility of confusion
6. \( *A \) indicates the item at address \( A \)
   - \( A \) may be the name of an array OR any address value <-- possibility of confusion
7. An address value can be an array name, a pointer variable, an integer, or any combination thereof

For each line of this program, explain which of the seven facts applies. [brack-stars.c]:
class 5: pointers

{ char a[8]; /* an array */
   char *p; /* a pointer */

   printf("a is at %u\n", a);
   strcpy(a, "abcdefg");
   p = a;
   printf("a contains [%s] p=%u, p points to [%s]\n", a, p, p);

   /* use brackets */
   a[2] = 'X';
   p = a + 2;
   p[1] = 'Y';
   p++;
   printf("a contains [%s] p=%u, p points to [%s]\n", a, p, p);

   /* use stars */
   *a = 'Z';
   *(a+4) = 'T';
   *p = 'A';
   *(p+4) = 'Q';
   printf("a contains [%s] p=%u, p points to [%s]\n", a, p, p);
}

Using [] to specify an item by position is called indexing, and using * to specify an item by its address is called pointing or dereferencing. When you work with arrays and pointers, you can use either method. Using pointers can be faster.

For a quick review, predict the output of [ptr-puzzle1.c] this code:

```c
int x[10] = { 2, 3, 4, 5, -1 } ; /* an array of 10 elements, 5 preset */
int *ip; /* a pointer to an int */
int i; /* a boring old int */
ip = x; /* ip points to the array */
i = (ip+5)[-2];
printf("%d %d %d %d %d", x[2], ip[ip[1]], *(ip+3), i);
```

Another review problem is [ptr-puzzle2.c]:

```c
main()
{
   int a[] = { 1, 4, 9, 16, 0 };
   int b[5];
   int *p;

   p = b;

   *p = a[2];
p[1] = *a;
*(p+3) = a[1];
p++;
*p = a[4];
b[1] += *(a+3);
printf("%d %d %d %d %d", b[0], b[1], b[2], *(b+3), *(b+4));
}
```

What is stored in b when this program is done?

13. Arrays of Pointers

Pointers are a basic type in C, just as ints, chars, and floats are basic types. You can create arrays of pointers. Consider this code: [array-of-pointers.c]:

```c
/*
 * array-of-pointers.c
 *
 * show how to define and use an array of pointers
 */
```
```c
main()
{
    int *a[5]; /* an array of five pointers to int */
    int x = 4; /* an actual int */
    int i;
    for(i=0; i<5; i++)
        a[i] = &x;
    for(i = 0; i<5; i++)
        printf("item %d points to location %d contains %d\n", i, a[i], *a[i]);
    *a[4] = 12;
    for(i = 0; i<5; i++)
        printf("item %d points to location %d contains %d\n", i, a[i], *a[i]);
}
```

The picture is:

```
   a
  /|
 / |
 x
```

Notice the syntax to create an array of pointers: `int *a[5]` The brackets at the end say we are creating an array of five elements, and the `int *` at the beginning says the elements in the array are pointers to int.

### 14. Pointers to Functions

The machine code for functions you write, like the data you enter, is stored in memory. These functions have addresses. C allows you to create and use pointers to functions. That is, you can use `&` to find the address of a function in memory, and you can pass those addresses to other functions. You can store these addresses in pointers, dereference these pointers to functions, even return them from functions. See later handouts for details.

### 15. Summary

The main ideas of this class are:

**Basic Definitions and Ideas**

1. A pointer variable is a variable that holds an address
2. `&` and `*`: `&X` is the address of X, `*P` is the thing pointed to by P

**Arrays and Pointers**

3. The name of an array is the address of the first element
   (if a is an array, then `a == &a[0]`)
4. An array is a sequence of elements, a pointer is a single value
5. `[]` and `*` may be used with array names and with pointers
6. `++/--` may be used with pointers but not with array names
7. `+-` may be used with pointers and with array names

**Pointing vs Indexing**

8. `x[i] == *(x+i)`
9. Using Pointers to Traverse Arrays: `*p++` is a central idea

**Pointers to Structs**
10. Pointers to structs are easy and useful
   struct time t;
   struct time *p;
   p = &t;

11. dot vs arrow: use dot for structs, arrow for pointers to structs
   p->hr = 12;
   t.min = 18;

**Arrays of Pointers**

12. An array of pointers is defined as *type a[len]*;
   char *cptr[LEN];
   int *ipt[LEN];