CS113: Lecture 6

Topics:

- Arrays
- Pointers and Arrays
- Pointer arithmetic
- Function pointers
- Strings
Arrays

- Often, programs use homogeneous data. For example, if we want to manipulate some grades, we might declare
  
  ```
  int grade0, grade1, grade2, grade3;
  ```

- If we have a large number of grades, it becomes cumbersome to represent/manipulate the grades, when each grade has a unique identifier. (How to find average? maximum? etc.)

- Arrays (a feature of many programming languages) allow us to refer to a number of instances of the same data type, using a single name.

- For example,
  ```
  int grade[4];
  ```
  makes available the use of int variables grade[0], grade[1], grade[2], grade[3], in a program.

- Note that arrays in C are zero-indexed – numbering begins at zero. If the size of an array `a` is `SIZE`, then the first accessible element of `a` is `a[0]`, and the last is `a[SIZE - 1]`.

- Now, to access elements of this array, we can write `grade[expr]`, where `expr` is any expression (evaluating to an integer within the appropriate range).
#include <stdio.h>

void main() {
    int grades[11], num_grades = 0;
    int i;
    float sum, average;

    printf("Please enter up to 10 grades, "
           "terminated by 0.\n" );
    scanf("%d", &(grades[num_grades]));
    while( grades[num_grades] != 0 ) {
        num_grades++;
        scanf("%d", &(grades[num_grades]));
    }

    /* Compute average */
    sum = 0;
    for( i = 0; i < num_grades; i++ ) {
        sum += grades[i];
    }
    /* Assume more than one grade entered */
    average = sum / num_grades;
    printf("The average of the grades is: \%f", average);
}
Arrays Are Also Pointers!

Pointers and arrays are almost exactly the same.

```c
void main() {
    int a[3];
    a[0] = 4;
    a[1] = 5;
    a[2] = 6;
}
```

- The type of `a` is `(int *)`.
- As a pointer, `a` points to the first memory location in the array.
- Stay tuned for details...
Arrays in C

- No bounds checking. Make sure that you only access array elements 0 through $N - 1$ for an array of size $N$.

A program that writes to “out-of-bounds” locations will compile and often run – beware! Writing to such invalid locations corrupts memory, sometimes the values of other variables. Very bad!

- The size of an array must be a constant. Here, “constant” means that the value can be determined at compile-time (so we know how much space to allocate in the activation record for the function).

```c
void func( int size ) {
    int b[size]; /* illegal */
    int g[(8 * 5) + 2]; /* fine */
}
```

- C has no internal mechanism for copying or comparing arrays.

If $a$, $b$ are arrays of the same type:

- expression $a = b$ is illegal – a declared array name cannot be treated as a variable, as $a$ is here.

- expression $a == b$ is legal – it checks to see if the pointers $a$ and $b$ point to the same memory location, and will return 0 (FALSE) if $a$ and $b$ are two different arrays with the same elements in them.
Example: Change-and-sum

```c
#include <stdio.h>

int change_and_sum( int *a, int size ) {
    int i, sum = 0;
    a[0] = 100;
    for( i = 0; i < size; i++ )
        sum += a[i];
    return sum;
}

void main() {
    int a[5] = { 0, 1, 2, 3, 4 };
    printf( "Sum of elements of a: %d\n", change_and_sum( a, 5 ) );
    printf( "Value of a[0]: %d\n", a[0] );
}

Notice:

- The shortcut to initialize the array
- Array passed as parameter – along with the size
- Function change_and_sum takes a pointer, and then treats it like an array
- Changes made to array persist!
```
Example: sorting numbers

```c
void sort_ints( int *a, int size ) {
    int i, j, k, temp;
    for( i = 0; i < size; i++ ) {
        /* find largest elt. of
        a[i], ..., a[size-1] */
        k = i;
        for( j = i + 1; j < size; j++ )
            if( a[j] > a[k] ) k = j;

        /* swap a[i], a[k] */
        temp = a[k];
        a[k] = a[i];
        a[i] = temp;
    }
}

void main() {
    int a[6] = { 3, 2, 8, 1, 5, 9 }, i;

    sort_ints( a, 6 );
    for( i = 0; i < 6; i++ )
        printf( "%d\n", a[i] );
}
```
More on pointers and arrays

- Suppose that \( a \) is an int array of size 10.

- If \( \text{pa} \) is a pointer to an integer, i.e.,
  
  ```
  int *\text{pa};
  ```
  
  then the assignment
  
  ```
  \text{pa} = &a[0];
  ```
  
  sets \( \text{pa} \) to point to element zero of \( a \).

- When does \( x = *\text{pa}; \) make sense – what does the type of \( x \) have to be? What does it do?

- If \( \text{pa} \) points to an element of an array, then (by definition) \( \text{pa} + 1 \) points to the next element. In general, \( \text{pa} + i \) points to the \( i \)th element after the element pointed to by \( \text{pa} \).

- Example.

```
int a[4] = { 0, 1, 2, 3 };
int *p;
p = &a[0];
printf( "\%d\n", *(p + 2));
scanf( "\%d", p + 3 );
printf( "You typed: \%d\n", a[3] );
```
Even more on pointers and arrays

• In fact, the name of an array is a synonym for the address of the initial element. As an example, when we have the declarations

```c
int a[10];
int *pa;
```

`&a[0]` is the same as `a`, and thus `pa = &a[0];` is the same as `pa = a;`.

• This is why the changes to an array made by a function persist: we were simply passing in a pointer to the first (zero indexed) element of the array.

• Accordingly, for any expression `b` of type `int *`, `b[i]` can always be written as `*(b + i)`, and vice-versa.

  For example, given the above declarations:
  `a[i]` and `*(a + i)` are equivalent
  `pa[i]` and `*(pa + i)` are equivalent

• Note that an array name (like `a` assuming the above declarations) is *not* a variable, so statements like `a = pa;` and `a++;` are illegal. (You also don’t want to form the expression `&a`.)
Practice: Pointers and Arrays

```c
void main() {
    int a[4] = { 0, 1, 2, 3 };  
    int *pa;

    pa = a + 1;
    printf( "%d\n", *pa );
    printf( "%d\n", pa[2] );
    pa++;
    printf( "%d\n", pa[0] );
    scanf( "%d", pa + 1 );
    printf( "You typed: %d\n", a[3] );
}
```
Pointer Arithmetic

• Pointer addition: pointer plus int

Saw that if a pointer \( p \) points to an element of an array, then \( p + i \) is a pointer (of the same type) pointing to the \( i \)th element after the element pointed to by \( p \).

• Pointer subtraction: pointer minus pointer

If \( p \) and \( q \) point to elements of the same array, then \( q - p \) gives the number of elements between \( p \) and \( q \).

• Pointer comparison: pointer relation pointer

Permissible relations: \( ==, !=, <, <=, >, >= \)

If \( p \) and \( q \) point to elements of the same array, then \( p < q \)
is true if \( p \) points to an earlier member of the array than \( q \) does.

• Note: CAN'T add two pointers, or perform any sort of multiplication, etc.

  – A pointer is a physical memory location, represented by an integer, but you should never think of them as integers. (Try it!).

  – Pointer arithmetic works at the level of “the next element in the array”, \( \text{NOT} \) at “the next physical memory address”.

11
Example: Elements before zero

(Example from PCP)

```c
void main() {
    int array[] = { 4, 5, 8, 9, 0, 1, 3, 2 }; 
    int index;

    index = 0;
    while( array[index] != 0 )
        index++;

    printf( "Number of elements before 0: %d\n", index );
}

void main() {
    int array[] = { 4, 5, 8, 9, 0, 1, 3, 2 }; 
    int *array_ptr;

    array_ptr = array;
    while(( *array_ptr ) != 0 )
        array_ptr++;

    printf( "Number of elements before 0: %d\n", array_ptr - array );
}
```
Function pointers

- Compiled code is just a bunch of 1’s and 0’s interpreted by the computer as instructions. Functions are just chunks of compiled code; they are stored at fixed memory locations and the chunks are copied onto the execution stack (as activation records) when the functions are invoked.

- Function pointers hold the addresses of functions; dereferencing a function pointer invokes the function. Function pointers can only point to functions that have the same kind of prototype.

- Declare a function pointer by writing a prototype for the kind of function you it to be able to point to, then add a * and parentheses, e.g.:

  ```
  int (*fctnptr)(int, int);
  ```

  declares `fctnptr` to be a pointer to a function that takes two ints as arguments and returns an int.

- Use the & operator applied to a function’s name to get the address of a function; use the * operator to dereference and invoke the function pointer.

- Function pointers can be placed in arrays, be used with typecasts, and follow the normal rules of pointer arithmetic.
Example: Function pointers

/* Function prototypes for functions defined later in the source file */
int add( int x, int y );
int subtract( int x, int y );
int multiply( int x, int y );

void main() {
    int (*ptr)( int, int );
    int a = 3, b = 4;

    ptr = &add;

    /* Prints 7 */
    printf( "Add: %d + %d = %d\n", a, b, (*ptr)(a,b) );

    ptr = &multiply;

    /* Prints 12 */
    printf( "Multiply: %d * %d = %d\n", a, b, (*ptr)(a,b) );
}

14
Example: Arrays of function pointers

/* Function prototypes for functions defined later in the source file */
int add( int x, int y );
int subtract( int x, int y );
int multiply( int x, int y );

void main() {
    int (*ptr)( int, int )[3];
    int a = 3, b = 4;

    ptr[0] = &add;
    ptr[1] = &subtract;
    ptr[2] = &multiply;

    /* Prints 7 */
    printf( "Add: %d + %d = %d\n", a, b, (*ptr[0])(a,b));

    /* Prints 12 */
    printf( "Multiply: %d * %d = %d\n", a, b, (*ptr[2])(a,b));
}
Strings: they’re just arrays!

• Strings are one-dimensional arrays of `char`s.

• By convention, a string in C is terminated by the null character, ‘\0’, or 0. (We have ‘\0’ == 0.)

• String constants (such as those passed to the function `printf`) are enclosed in double quotes.

• When allocating `char` arrays that will hold strings, make sure you allocate enough space!
  
  – When dealing with strings in C, you should always think of the underlying array of characters.

  – Also: always think in terms of the activation records! You must explicitly allocate all the space for every string you use, and space allocated as part of a function call will be destroyed when the function finishes.

  – We’re not in Java anymore. Are you starting to miss it?
Example: “Double” printing

#include <stdio.h>

void dprint( char *s ) {
    int i;
    /* for this loop to exit, s
     * better terminate with 0! */
    for( i = 0; s[i] != 0; i++ )
        printf( "%c%c", s[i], s[i] );
}

void main() {
    /* s and s2 are the same strings */
    char s[] = "Hi!";
    char s2[] = { 'H', 'i', '!', '\0' };

    dprint( s ); /* HHii!! */
    if( s == s2 ) {
        printf("Points to identical string");
    } else {
        printf("Does not");
    }
}
Example: “squeeze” function

(Based on an example from K&R)

#include <stdio.h>

/* squeeze deletes all instances of the character c from the string s. */
void squeeze( char *s, int c ) {
    int i, j;
    
    for( i = j = 0; s[i] != 0; i++ ) {
        if( s[i] != c ) {
            s[j] = s[i];
            j++;
        }
    }
    s[j] = 0;
}

void main() {
    char s[100];
    strcpy( s, "Clzzeazn mez zup!" );
    printf( "Before squeeze: %s\n", s );
    squeeze( s, 'z' );
    printf( "After squeeze: %s\n", s );
}
String handling functions

These are from string.h. See Appendix B3 of K&R for an exhaustive list.

- \textbf{int strlen( char *s )};
  Returns the length of the string s.

- \textbf{char *strcat( char *s1, char *s2 )};
  Takes two strings as arguments, concatenates them, and puts the result in s1. The programmer must ensure that s1 points to enough space to hold the result. The string s1 is returned.

- \textbf{char *strcpy( char *s1, char *s2 )};
  The string s2 is copied into s1. Whatever exists in s1 is overwritten. It is assumed that s1 has enough space to hold the result. The value of s1 is returned.
  (Remember, using = to assign one string to another only copies pointers, it doesn't actually give a new copy of the string. And it won't work at all if the left hand side is a string array.)

- \textbf{int strcmp( char *s1, char *s2 )};
  Integer is returned that is less than, equal to, or greater than zero, depending on whether s1 is lexicographically less than, equal to, or greater than s2 (respectively).

\textit{A good exercise is to implement these functions yourself.}
The strcmp ordering: think dictionary

From “lowest” to “highest”:

"1"
"128"
"16"
"2"
"32"
"4"
"64"
"8"
"Avocado"
"Can"
"Can not"
"Can’t"
"Cannot"
"Cantor"
"Lime"
"apple"
"banana"
"c"
"c language"
"c programmer"
"cantaloupe"
Example: Reversing a string

```c
#include <string.h>

void reverse( char *s ) {
    int halflen, len, i;
    char temp;

    len = strlen( s );
    halflen = len / 2;

    for( i = 0; i < halflen; i++ ) {
        /* swap s[i] and s[len - 1 - i] */
        temp = s[i];
        s[i] = s[len - 1 - i];
        s[len - 1 - i] = temp;
    }
}

void main() {
    char s[20];
    strcpy( s, "\.desrever ma I" );
    printf( "Before reversal: %s\n", s );
    reverse( s );
    printf( "After reversal: %s\n", s );
}
```
Multidimensional arrays

- Arrays can have more than one dimension.

- Example of declaring a two-dimensional array of ints:
  ```
  int b[3][7];
  ```
  Makes available 21 ints for use: \( b[i][j] \) where \( i \) ranges from 0 to 2, and \( j \) ranges from 0 to 6.

- Can also declare three-dimensional, etc. arrays.
  ```
  int c[2][4][10];
  ```
Arrays of Strings

void get_string( char s[] ) {
    scanf( "%s", s );
    printf( "Length of your string: " );
    printf( "%d\n", strlen( s ) );
}

void main() {
    char arr[8][81];
    get_string( arr[1] );
    printf( "You typed the string: %s\n", arr[1] );
    printf( "The first character you typed was: " );
    printf( "%c\n", arr[1][0] );
}

Notice:

- Two-dimensional array of chars acts as array of strings (of size 8): arr[0], ..., arr[7]

- scanf( "%s", ... ); used to read strings. (It’s dangerous – we’ll see a better next time.)

- To refer to a specific character of one of the strings arr[i], tack on another index: arr[1][0] for instance refers to the first (zero-indexed) character of the string arr[1]
strlen implementations

(from K&R)

int strlen( char *s ) {
    int n;
    for( n = 0; *s != '\0'; s++ )
        n++;
    return n;
}

/* pointer arithmetic version */
int strlen( char *s ) {
    char *p = s;
    while( *p != '\0' ) p++;
    return( p - s );
}
strcpy implementations

/* "obvious" way */
void strcpy( char *s, char *t ) {
    int i = 0;

    do {
        s[i] = t[i];
    } while (t[i++] != '\0');
}

/* slick pointer version */
void strcpy( char *s, char *t ) {
    while( *s++ = *t++ );
}
What does \verb|==| do here?

```c
void main() {
    char s[20];
    strcpy( s, "Hello" );
    if( s == "Hello" ) {
        printf( "Equal.\n" );
    } else {
        printf( "Not equal.\n" );
    }
}
```

What's going on?

- A **string literal** like "Hello" is represented in the function's activation record as a `char` array, like a regular string. (It has to go somewhere, right?!)  

- BUT the array is static (which we'll talk about later) AND it can't be modified.

- Thus `s = "Hello"`; doesn't behave as you might think.

- Are the two instances of "Hello" in the function above stored in distinct arrays? It's implementation-dependent.