Announcements

- Anyone have any specific issues with c in the lab so far ??

- Have link to setup eclipse, will post on resource section and demo in lab wednesday
Arrays

- An array is a group of memory locations with the same name and type

- To get to a particular element in the array, c needs to know about
  - data type
  - name
  - Length or position

- Array length can be determined:
  - statically— at compile time (when we code)
    - e.g., char str1[10];
  - dynamically— at run time (c can create memory on the fly for you to use)
    - e.g., char *str2;
Defining a variable is called “allocating memory” to store that variable.

Defining an array means allocating memory for a group of bytes.

Individual array elements are indexed:
- 0..n-1
- [location]
- Initializing the arrays are your problem
  `int a[3];`
  `...`
  `X = a[1];`  `...`

- Bound checking is your problem
  `printf("%d",a[100]);`  `...`
We can say for example:

```c
int varX = C[4] / 5;
```

Declarations:

```c
int b[100], v[3];
```
More arrays

- Can also create arrays in the following manners
  1. `int a[] = {1,2,3};`
  2. `int b[3] = {6,3,7};`
  3. `int n[10] = {0};`

- In general: you need to initialize the array elements
  - 3 is a trick case.
Command Line Args

int main( int argc, char *argv[] )

- argc is the argument count
- argv is the argument vector
  - array of strings with command-line arguments
- the int value is the return value
  - convention: return value of 0 means success,
    - > 0 means there was some kind of error
  - can also declare as void (no return value)
Library

- Access libraries using the `include` statement
- Generally include header files
  - Your own `.c` files will include `.h`
  - `.h` NEVER includes `.c`
  - Tell makefile about all `.c` files
- Compiler links them automatically
- Example:
  - Standard input/output: `stdio.h`
  - To look up information use the `man` page:
    
    ```bash
    man stdio
    ```
NAME

stdio - standard input/output library functions

SYNOPSIS

#include <stdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;

DESCRIPTION

The standard I/O library provides a simple and efficient buffered stream I/O interface. Input and output is mapped into logical data streams and the physical I/O characteristics are concealed. The functions and macros are listed below; more information is available from the individual man pages.

A stream is associated with an external file (which may be a physical device) by opening a file, which may involve creating a new file. Creating an existing file causes its former contents to be discarded. If a file can support positioning requests (such as a disk file, as opposed to a terminal) then a file position indicator associated with
stdio.h

- Access stdio functions by
  - using #include <stdio.h>
  - compiler links it automatically

- defines stdin, stdout, stderr
- use for character, string and file I/O (later)

- Example: printf
printf Function

- The way printf works is it takes a format to print out and then the data to add to the format.

- One or more of the following:
  - `%[flags][width][.precision][modifiers]type`
    - “%d”
      - Means a single number
    - “%d %d %d”
      - ??
printf ("%d %d", a, b);
**stdio.h : printf, type specifier**

- `int printf(const char *format, ...)`: formatted output to stdout

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Character</td>
<td>a</td>
</tr>
<tr>
<td>d or i</td>
<td>Signed decimal integer</td>
<td>392</td>
</tr>
<tr>
<td>e</td>
<td>Scientific notation (mantissa/exponent) using e character</td>
<td>3.9265e2</td>
</tr>
<tr>
<td>E</td>
<td>Scientific notation (mantissa/exponent) using E character</td>
<td>3.9265E2</td>
</tr>
<tr>
<td>f</td>
<td>Decimal floating point</td>
<td>392.65</td>
</tr>
<tr>
<td>g</td>
<td>Use shorter %e or %f</td>
<td>392.65</td>
</tr>
<tr>
<td>G</td>
<td>Use shorter %E or %f</td>
<td>392.65</td>
</tr>
<tr>
<td>o</td>
<td>Signed octal</td>
<td>610</td>
</tr>
<tr>
<td>s</td>
<td>String of characters</td>
<td>sample</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal integer</td>
<td>7235</td>
</tr>
<tr>
<td>x</td>
<td>Unsigned hexadecimal integer</td>
<td>7fa</td>
</tr>
<tr>
<td>X</td>
<td>Unsigned hexadecimal integer (capital letters)</td>
<td>7FA</td>
</tr>
<tr>
<td>p</td>
<td>Address pointed by the argument</td>
<td>B800:0000</td>
</tr>
<tr>
<td>n</td>
<td>Nothing printed. The argument must be a pointer to integer where the number of characters written so far will be stored.</td>
<td></td>
</tr>
</tbody>
</table>
### printf flags

- `%[flags][width][.precision][modifiers]type`

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-</code></td>
<td>Left align within the given width. (right align is the default).</td>
</tr>
<tr>
<td><code>+</code></td>
<td>Forces to preceed the result with a sign (+ or -) if signed type. (by default only - (minus) is printed).</td>
</tr>
<tr>
<td>Blank</td>
<td>If the argument is a positive signed value, a blank is inserted before the number.</td>
</tr>
<tr>
<td><code>#</code></td>
<td>Used with o, x or X type the value is preceeded with 0, 0x or 0X respectively if non-zero.</td>
</tr>
<tr>
<td></td>
<td>Used with e, E or f forces the output value to contain a decimal point even if only zeros follow.</td>
</tr>
<tr>
<td></td>
<td>Used with g or G the result is the same as e or E but trailing zeros are not removed.</td>
</tr>
</tbody>
</table>
int class_size = 35;
char class_name[15] = “3157 adv prog”; 

printf(“Welcome to our test program\n”);

printf( “the %s class size is %d”,
        class_name, class_size);
int array

1. #include <stdio.h>
2. #define MAX 6
3. int main( void ) {
4.   int arr[MAX] = { -45, 6, 0, 72, 1543, 62 };
5.   int i;
6.   for ( i=0; i<MAX; i++ ) {
7.     printf( "[%d] = %d \n", i, arr[i] );
8.   }
9. } /* end of main() */
stdio.h: scanf

- int scanf(const char *format, ...)

Example: scanf/printf

```c
#include <stdio.h>
void main( void ) {
    int n = 0; /* initialization required */

    printf( "how much wood could a woodchuck chuck\n"");
    printf( "if a woodchuck could chuck wood?" ); /*
         prompt user */

    scanf( "%d",&n ); /* read input */

    printf( "the woodchuck can chuck %d pieces of
         wood!\n",n );
    return;
}
```
output

$ a.out
how much wood could a woodchuck chuck
if a woodchuck could chuck wood? 12345
the woodchuck can chuck 12345 pieces of
wood!
Loops

- loops in C are just like in Java

- there are 2 methods for looping:
  - counter-controlled (loop for a fixed number of times)
  - sentinel-controlled (loop while a condition is true)

- there are 3 statements for implementing the 2 methodologies:
  - for
  - while
  - do...while

- as always: beware the infinite loop!

- Ctrl-C interrupts your executing C program
Branching

- branching in C is just like in Java

- there are 2 ways to do branching:
  - if/else
  - switch

- questions:
  - which is more flexible and powerful?
  - one can always be translated into the other, but not the other way around— which is which?
WARNING!!!!
Pointers

- Unique to c/c++
- Lots of confusion
- Please ask if something doesn’t make sense
Pointer power

- Variables that contain memory addresses as their values
- Data types we’ve learned about in C use direct addressing
- Pointers facilitate indirect addressing

- Declaring pointers:
  - pointers indirectly address memory where data of the types we’ve already discussed is stored (e.g., int, char, float, etc.)
  - declaration uses asterisks (*) to indicate a pointer to a memory location storing a particular data type
  - Called dereferencing a pointer

- example:
  int *count;
  float *avg;
Pointers: nitty gritty

- ampersand & is used to get the address of a variable location (dereference a pointer)

- example:
  ```c
  int count = 12;
  int *countPtr = &count;
  ``
  
  &count returns the address of count and stores it in the pointer variable countPtr
Another example

- here’s another example:
  ```c
  int i = 3, j = -99;
  int count = 12;
  int *countPtr = &count;
  printf ( "%d", *countPtr);
  ```

- Here is the memory picture:
Arrays as pointers

- an array is some number of contiguous memory locations
- an array definition is really a pointer to the starting memory location of the array
- and pointers are really integers
- so you can perform integer arithmetic on them
- e.g., +1 increments a pointer, -1 decrements
- you can use this to move from one array element to another
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int i, *j, arr[5];
srand( time ( NULL ));
    for ( i=0; i<5; i++ )
        arr[i] = rand() % 100;
    printf( "arr=%p\n",arr );
    for ( i=0; i<5; i++ ) {
        printf( "i=%d arr[i]=%d &arr[i]=%p\n",i,arr[i],&arr[i] );
    }
    j = &arr[0];
    printf( "\nj=%p *j=%d\n",j,*j );
    j++;
    printf( "after adding 1 to j:\n j=%p *j=%d\n",j,*j );
}
```
Output

arr=0xbffffff4f0
i=0 arr[i]=29 &arr[i]=0xbffffff4f0
i=1 arr[i]=8  &arr[i]=0xbffffff4f4
i=2 arr[i]=18 &arr[i]=0xbffffff4f8
i=3 arr[i]=95 &arr[i]=0xbffffff4fc
i=4 arr[i]=48 &arr[i]=0xbffffff500
j=0xbffffff4f0  *j=29
after adding 1 to j:
j=0xbffffff4f4  *j=8
Pointer operations

- HUGE Difference between
  - `ptr++`
  - `*ptr++`

- `int b[5] ....`
  `int *bPtr;`

  `bPtr = b`       // or
  `bPtr = &b[0]`
Careful when moving pointers

bPTr += 2;

the memory location isn’t simply incremented by 2.....depends on size of type being pointed to.
Strings

- Used in last lab
  - storing multiple characters as array
  - data type is still char
- BUT it has a length
  - Max length
  - Current length (till the \0)
  - last character the is terminator: ‘\0’, aka NULL
  - string constants are surrounded by double quotes: "

- example:
  - char s[6] = "ABCDE";
String II

- char s[6] = “ABCDE”;
- Memory storage looks like:

  A B C D E \0

- Need to remember that you are really accessing indices 0 – (length-2) since the value at length-1 is always \0

- Need to allocate enough memory
Using strings

- printing strings
- format sequence: %s
- example:

```c
#include <stdio.h>
int main() {
    char str[6] = "ABCDE";
    printf( "str = %s\n", str );
} /* end of main() */
```
String Library

- to use the string library, include the header in your C source file:
  ```c
  #include <string.h>
  ```
- string length function:
  ```c
  int strlen( char *str );
  ```
  this function returns the number of characters in `str`; note that this
  is NOT the same thing as the number of characters allocated for
  the string array ..... 
- string comparison function:
  ```c
  int strcmp( const char *s1, const char *s2 );
  ```
  “This function returns an integer greater than, equal to, or less
  than 0, if the string pointed to by `s1` is greater than, equal to, or
  less than the string pointed to by `s2` respectively. The sign of a
  non-zero return value is determined by the sign of the difference
  between the values of the first pair of bytes that differ in the
  strings being compared.”

man strcmp
copying functions:

char *strcpy( char *dest, char *source );
- copies characters from source array into dest array up to NULL
- Need to make sure you have space...

char *strncpy( char *dest, char *source, int num );
- copies characters from source array into dest array; stops after num characters (if no NULL before that); appends NUL
Search

char *strchr( const char *source, const char ch );

- returns pointer to first occurrence of ch in source; NULL if none

char *strstr( const char *source, const char *search );

- return pointer to first occurrence of search in source
String Parsing

`char *strtok( char *s1, const char *s2 );`

- breaks string `s1` into a series of tokens, delimited by `s2`
- called the first time with `s1` equal to the string you want to break up
- called subsequent times with NULL as the first argument
- each time is called, it returns the next token on the string
- returns null when no more tokens remain
Example

```c
char inputline[1024];
char *name, *rank, *serial_num;
printf( "enter name+rank+serial number: " );
scanf( "%s", inputline );
name = strtok( inputline,"+" );
rank = strtok( null,"+" );
serial_num = strtok( null,"+" );
```
Formatting functions

int sscanf(char *string, char *format, ...)
- parse the contents of string according to format
- placed the parsed items into 3rd, 4th, 5th, ... argument
- return the number of successful conversions

int sprintf(char *buffer, char *format, ...)
- produce a string formatted according to format
- place this string into the buffer
- the 3rd, 4th, 5th, ... arguments are formatted
- return number of successful conversions

- format characters are like printf and scanf (see notes from earlier lectures)
Memory allocations

- One of the most powerful features of C is the ability of the programmer to create more memory space during the execution of the program.
- Limited by physical machine memory
- If you want to be able to create memory, you also need to free it manually
`malloc /sizeof / free`

- `charPtr = malloc ( sizeof ( ... ) );`
- `free (charPtr)`
Memory allocations

- One of the most powerful features of C is the ability of the programmer to create more memory space during the execution of the program.
- Limited by physical machine memory
- If you want to be able to create memory, you also need to free it manually
- When you don’t it is called memory leaking...more on this later
Array vs memory allocation

- Arrays are great when you have a rough idea of how many items you will be dealing with:
  - 10 numbers
  - 30 students
  - Less than 256 characters of input
Map of memory

- Think of memory as a box
- Main is placed on the bottom and any variable on top of that
- Any function call gets placed on top of that
- This part of memory grows upward
- It is called the stack
- Your program is over when the stack is empty
heap

- The heap is the other side of memory
- Global variables, and allocated memory is created on the heap
- It grows downwards
HEAP

STACK
Dynamic Memory Allocation

- pre-allocated memory comes from the “stack”
- dynamically allocated memory comes from the “heap”
- To get memory you allocated (malloc) memory, and to let it go, you free it (free)
- family of functions in stdlib, including:

  ```c
  void *malloc( size_t size );
  void *realloc( void *ptr, size_t size );
  void free( void * );
  ```
malloc and realloc return a generic pointer (void *) and you have to “cast” the return to the type of pointer you want.

That is if you are allocation a bunch of characters, you say

Ptr = (char*) malloc....
#include <stdio.h>
#include <stdlib.h>
#define BLKSIZ 10

main() {
    FILE *fp;
    char *buf, k;
    int bufsiz, i;
    // open file for reading
    if (( fp = fopen( "myfile.dat","r" )) == NULL ) {
        perror( "error opening myfile.dat" );
        exit( 1 );
    }
    // allocate memory for input buffer
    bufsiz = BLKSIZ;
    buf = (char *)malloc( sizeof(char)*bufsiz );
// read contents of file
    i = 0;
    while (( k = fgetc( fp )) != EOF ) {
        buf[i++] = k;
        if ( i == bufsiz ) {
            bufsiz += BLKSIZ;
            buf = (char *)realloc( buf,sizeof(char)*bufsiz );
        }
    }
    if ( i >= bufsiz-1 ) {
        bufsiz += BLKSIZ;
        buf = (char *)realloc( buf,sizeof(char)*bufsiz );
    }
    buf[i] = '\0';
// output file contents to the screen
    printf( "buf=[%s]\n",buf );
// close file
    fclose( fp );
} // end main()
Dynamic memory

- `malloc()` allocates a block of memory:
  ```c
  void *malloc( size_t size );
  ```
- Lifetime of the block is until memory is freed, with `free()`:
  ```c
  void free( void *ptr );
  ```

- Example:
  ```c
  int *dynvec, num_elements;
  printf( "how many elements do you want to enter? " );
  scanf( "%d", &num_elements );
  dynvec = (int *)malloc( sizeof(int) * num_elements );
  ```