Announcements

- Hope you have started the lab
- Includes c background reading

- Posted rough schedule till semester end
Outline

- In order to complete the lab, you will need to be pointer masters
  - Will cover all kinds today
  - Will do some short examples
  - Get to wear pointy ‘c master’ hat if you finish the lab on time 😊
Project 2

- Will be posted tonight

- Link organizer program written in c

- Will allow you to add, delete, and view links for webpages
  - Written in c, runs off the cs webservers

- Will hook into perl program which can generate html snapshots so that you can show ‘thumbnail’ next to each link
  - Will learn how to communicate between perl and c
  - Will learn how to communicate between solaris and linux to pass back and forth data
  - Will cover this in labs so homework will be easier
  - Will get off one class to cover this (apr 4 and apr 9).
Short Review

Let’s compare in C:

- int a;
- int b[10];
- int c[] = “apple”;  
- int *d;
Strings

- when you use double quotes, C will treat it as a bunch of characters followed by end slash
- so "\0" is really slash, zero, end slash
  - single quote ‘\0’ is what you probably meant

- char *n = “something”;
  - this will do what you are trying to do in the short run
  - will mess everything up in the long run
  - WHY???
Example

- `char *a1 = "testing";`
- `char a2[] = "testing";`

- what is the difference ??

- note: many people meant to do `strcpy(a1,"testing");`
Formatting functions

```c
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

```c
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)
Reminder

- `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

```c
char *ptr = (char*) malloc( sizeof(char) * 12);
```
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 );
  ```
Memory leaking

- memory leaks—memory allocated that is never freed:

```c
char *combine( char *s, char *t ) {
    u = (char *)malloc( strlen(s) + strlen(t) + 1 );
    if ( s != t ) {
        strcpy( u, s );
        strcat( u, t );
        return u;
    }
    else {
        return 0;
    }
} /* end of combine() */
```

- u should be freed if return 0; is executed
- but you don’t need to free it if you are still using it!
Example 2

```c
int main(void) {

    char *string1 = (char*)malloc(sizeof(char)*50);
    char *string2 = (char*)malloc(sizeof(char)*50);
    scanf("%s",string2);
    string1 = strong2;  //what is the problem here ??
    ...
    free(string2);
    free(string1);  //???

    return 0
}
```
Memory leak tools

- Purify
- Valgrind
- Insure++
- Memwatch (will use it in lab)
- Memtrace
- Dmalloc
Dynamic memory

- note: malloc() does not initialize data, that is you have garbage there with whatever was there in memory
- you can allocate and initialize with “calloc”:
  ```c
  void *calloc( size_t nmemb, size_t size );
  ```
  - calloc allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero.

- you can also change size of allocated memory blocks with “realloc”:
  ```c
  void *realloc( void *ptr, size_t size );
  ```
  - realloc changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized.

- these are all functions in stdlib.h
- for more information: man malloc
Dynamic arrays

- “arrays” are defined by specifying an element type and number of elements
  - statically:
    int vec[100];
    char str[30];
    float m[10][10];
  - dynamically:
    int * dynvec;
    int num_elements;
    printf("how many elements do you want to enter? ");
    scanf("%d", &num_elements);
    dynvec = (int *) malloc( sizeof(int) * num_elements );

- for an array containing N elements, indices are 0..N-1
- stored as a linear arrangement of elements
- often similar to pointers
Dynamic arrays II

- C does not remember how large arrays are (i.e., no length attribute, unlike Java)

- given:
  ```c
  int x[10];
  x[10] = 5; /* error that compile will not notice! */
  ```

- ERROR! because you have only defined x[0]..x[9] and the memory location where x[10] is can become something else...

- sizeof x gives the number of bytes in the array
- sizeof x[0] gives the number of bytes in one array element
- You can compute the length of x via:

  ```c
  int length_x = sizeof x / sizeof x[0];
  ```
Arrays cont.

- when an array is passed as a parameter to a function:
  - The size information is not available inside the function, since you are only passing in a start memory location
  - When programming C, array size is typically passed as an additional parameter

```c
printArray( x, length_x );
```

- or globally

```c
#define VECSIZE 10
int x[VECSIZE];
```
Dynamically allocated arrays

- C references arrays by the address of their first element
- Name of array is equivalent to &array[0]
- you can iterate through arrays using pointers as well as indexes:

```c
int *v, *last;
int sum = 0;
last = &x[length_x-1];
for ( v = x; v <= last; v++ )
    sum += *v;
```
2 dimensional arrays

- 2-dimensional arrays
  - This is much different than double arrays in java
- `int weekends[52][2];`
- you can use indices or pointer math to locate elements in the array
  - `weekends[0][1]`
  - `weekends+1`
- `weekends[2][1]` is same as `*(weekends+2*2+1)`,
- but NOT the same as `*weekends+2*2+1`
  - WHY??
double arrays

- char multi[2][10];
- multi[0] = {'0','1','2','3','4','5','6','7','8','9'}
- multi[1] = {'a','b','c','d','e','f','g','h','i','j'}

- in memory:
  - 0123456789abcdefhij

*(*(multi + row) + col)

multi[row][col]
passing in arrays

- when you pass in an array in your function
  
  - int func1(int *array, int size)
  - int func2(int array[], int size)
int (*p1d)[10]

is not the same as

int *p1d[10];
Difference

```c
void foo( int a, int b ) {
    int tmp = a;
    a = b;
    b = tmp;
} // end foo()

void foo2( int *a, int *b ) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
} // end foo2()
```
int main() {
    int number = 10;
    foo(&number);
    return 0;
}

void foo(int *p) {
    *p = 30;
}
**const**

- Keyword in c/c++
- Allows the compiler to know which values shouldn’t be modified
- Added into c later

**Example:**

```c
const int a = 5;

void foo(const int x) {
     
}
```
const

- Better than #define since error message will be easier to understand since preprocessor not involved
- Very useful in functions to either return const or make sure a pointer doesn’t alter the original object
Const pointer to non-const

- This is a pointer which always points to the same location, but the value can be modified.

```c
int * const ptr = &x;
```

```c
*ptr = ??
can’t say
ptr = & ??
```

- Example2: array name
Const pointer to const data

- `int x = 200;`
- `const int * const ptr = &x;`
four choices

- Some confusion
  - int *A
  
  - int const * X
    - const int * X
    //variable pointer to const
  
  - int * const Y
    //const pointer to int
  
  - int const * const Z
    //const point to const
Pointers to functions

- C allows you to also pass around a pointer to a function

```c
void foo (int , int (*)(int , int));
```

```c
int example1(int x, int y) { return x+y; }
```

```c
foo(5, example1);
```
void foo(int a, int (*A)(int, int)){

    if((A)(5,10) > 0){

    }

else {

}

}
note

- if you have a const variable and pointer doesn’t put any modifier (just plain pointer) can modify the variable through the pointer (some compilers will warn but compile)
switch gears
Compiling problems

- errors can come from multiple sources:
  - *pre-processor*: missing include files
  - *parser*: syntax errors
  - *assembler*: rare
  - *linker*: missing libraries and references
  - e.g., undefined names will be reported when linking:

```bash
undefined symbol first referenced in file
  _print program.o
  ld fatal: Symbol referencing errors
  No output written to file.
```

- if gcc gets confused, there can be hundreds of messages!
  - fix first message first, and then retry— ignore the rest

- gcc will produce an executable with warnings

- look at the errors carefully
  - check line numbers
Implicit conversions

- implicit:
  ```
  int a = 1;
  char b = 97; // converts int to char
  int s = a + b; // adds int and char, converts to int
  ```

- promotion: char -> short -> int -> float -> double
- if one operand is double, the other is made double
- else if either is float, the other is made float

```
int a = 3;
float x = 97.6;
double y = 145.987;
y = x * y; // x becomes double; result is double
x = x + a; // a becomes float; result is float
``` 

- real (float or double) to int truncates
explicit

- explicit:
- type casting
  ```
  int a = 3;
  float x = 97.6;
  double y = 145.987;
  y = (double)x * y;
  x = x + (float)a;
  ```
- using functions (in math library...)
  1. floor() – rounds to largest integer not greater than x
  2. ceil() - round to smallest integer not smaller than x
  3. round() – rounds up from halfway integer values
Example

#include <stdio.h>
#include <math.h>
int main() {
    int j, i, x;
    double f = 12.00;
    for ( j=0; j<10; j++ ) {
        i = f;
        x = (int)f;
        printf( "f=%.2f i=%d x=%d
floor(f)=%.2f ceil(f)=%.2f round(f)=%.2f\n", f, i, x, floor(f), ceil(f), round(f) );
        f += 0.10;
    } // end for j
} // end main()
Output

- $f=12.00$  $i=12$  $x=12$  $\text{floor}(f)=12.00$  $\text{ceil}(f)=12.00$  $\text{round}(f)=12.00$
- $f=12.10$  $i=12$  $x=12$  $\text{floor}(f)=12.00$  $\text{ceil}(f)=13.00$  $\text{round}(f)=12.00$
- $f=12.20$  $i=12$  $x=12$  $\text{floor}(f)=12.00$  $\text{ceil}(f)=13.00$  $\text{round}(f)=12.00$
- $f=12.30$  $i=12$  $x=12$  $\text{floor}(f)=12.00$  $\text{ceil}(f)=13.00$  $\text{round}(f)=12.00$
- $f=12.40$  $i=12$  $x=12$  $\text{floor}(f)=12.00$  $\text{ceil}(f)=13.00$  $\text{round}(f)=12.00$
- $f=12.50$  $i=12$  $x=12$  $\text{floor}(f)=12.00$  $\text{ceil}(f)=13.00$  $\text{round}(f)=12.00$
- $f=12.60$  $i=12$  $x=12$  $\text{floor}(f)=12.00$  $\text{ceil}(f)=13.00$  $\text{round}(f)=13.00$
- $f=12.70$  $i=12$  $x=12$  $\text{floor}(f)=12.00$  $\text{ceil}(f)=13.00$  $\text{round}(f)=13.00$
- $f=12.80$  $i=12$  $x=12$  $\text{floor}(f)=12.00$  $\text{ceil}(f)=13.00$  $\text{round}(f)=13.00$
- $f=12.90$  $i=12$  $x=12$  $\text{floor}(f)=12.00$  $\text{ceil}(f)=13.00$  $\text{round}(f)=13.00$
Be aware

- almost any conversion does something—but not necessarily what you intended!!

- example:
  
  ```c
  int x = 100000;
  short s = x;
  printf("%d %d\n", x, s);
  
  - output is:
    100000 -31072
  
  - WHY?
  ```
math library

- Functions ceil() and floor() come from the math library definitions:
  - ceil( x ): returns the smallest integer not less than x, as a double
  - floor( x ): returns the largest integer not greater than x, as a double

- In order to use these functions, you need to do two things:
  1. include the prototypes (i.e., function definitions) in the source code:
     ```
     #include <math.h>
     ```
  2. include the library (i.e., functions’ object code) at link time:
     ```
     unix$ gcc abcd.c -lm
     ```

- Exercise: can you write a program that rounds a floating point?
math

- some other functions from the math library (these are function prototypes):
  - double sqrt( double x );
  - double pow( double x, double y );
  - double exp( double x );
  - double log( double x );
  - double sin( double x );
  - double cos( double x );

- questions:
  - can you make sense of /usr/include/math.h?
  - where are the definitions of the above functions?
  - what are other math library functions?
Random numbers

- with computers, nothing is random (even though it may seem so at times...)

- there are two steps to using random numbers in C:
  1. seeding the random number generator
  2. generating random number(s)

- standard library function:
  ```c
  #include <stdlib.h>
  ```

- seed function:
  ```c
  srand( time ( NULL ));
  ```

- random number function returns a number between 0 and RAND_MAX (which is $2^{32}$)
  ```c
  int i = rand();
  ```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main( void ) {
    int r;
    srand( time ( NULL ));
    r = rand() % 100;
    printf( "pick a number between 0 and 100...
" );
    printf( "was %d your number?", r );
}
#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()
Character handling

- character handling library
  
  \[
  \text{#include <ctype.h>}
  \]

- digit recognition functions (bases 10 and 16)

- alphanumeric character recognition

- case recognition/conversion

- character type recognition

- these are all of the form:
  
  \[
  \text{int isdigit( int c );}
  \]

- where the argument \(c\) is declared as an int, but it is interpreted as a char

- so if \(c = '0'\) (i.e., the ASCII value ‘0’, index=48), then the function returns true (non-zero int)

  but if \(c = 0\) (i.e., the ASCII value NULL, index=0), then the function returns false (0)
digits

- digit recognition functions (bases 10 and 16)
  
  ```c
  int isdigit( int c );
  ```

  - returns true (i.e., non-zero int) if c is a decimal digit (i.e., in the range ’0’..’9’);
    returns 0 otherwise

  ```c
  int isxdigit( int c );
  ```

  - returns true (i.e., non-zero int) if c is a hexadecimal digit (i.e., in the range ’0’..’9’,’A’..’F’);
    returns 0 otherwise
Alpha numeric

- alphanumeric character recognition
  
  ```c
  int isalpha( int c );
  ```

- returns true (i.e., non-zero int) if `c` is a letter (i.e., in the range ‘A’..'Z','a'..'z’); returns 0 otherwise

```c
int isalnum( int c );
```

- returns true (i.e., non-zero int) if `c` is an alphanumeric character (i.e., in the range ‘A’..'Z','a'..'z','0'..'9’); returns 0 otherwise
Case

- **case recognition**
  - int islower( int c );
  - returns true (i.e., non-zero int) if c is a lowercase letter (i.e., in the range ‘a’..'z');
  - returns 0 otherwise

- int isupper( int c );
  - returns true (i.e., non-zero int) if c is an uppercase letter (i.e., in the range ‘A’..'Z');
  - returns 0 otherwise

- **case conversion**
  - int tolower( int c );
  - returns the value of c converted to a lowercase letter (does nothing if c is not a letter or if c is already lowercase)

- int toupper( int c );
  - returns the value of c converted to an uppercase letter (does nothing if c is not a letter or if c is already uppercase)
types

- character type recognition
  int isspace( int c );
  returns true (i.e., non-zero int) if c is a space; returns 0 otherwise

- int iscntrl( int c );
  returns true (i.e., non-zero int) if c is a control character; returns 0 otherwise

- int ispunct( int c );
  returns true (i.e., non-zero int) if c is a punctuation mark; returns 0 otherwise

- int isprint( int c );
  returns true (i.e., non-zero int) if c is a printable character; returns 0 otherwise

- int isgraph( int c );
  returns true (i.e., non-zero int) if c is a graphics character; returns 0 otherwise