CS3157: Advanced Programming

Lecture #7
Oct 23
Shlomo Hershkop
shlomo@cs.columbia.edu

Announcements

- Make sure you are covering the readings
- First 2 labs should be graded, Tas are working on homework grades
- lab are in order….if something is not clear, please ask
Today

- more basic c
  - pointer
  - memory allocation
  - string handling
  - stuff

- reading:
  - Memory management

Reminder: Split personalities

- In c and cpp normal to divide definition of code (header files .h) and working code (.c files)

- So will have function declaration (.h)
  - int foo();

- And function definitions (.c)
  - int foo(){. . . . . }
Header files

- .h files usually used to define methods or centralize definitions
- public int calculateSomething(int []);
- Can either name the variables or not
- int[] vs int ar[]
- In .c file use; #include “something.h”

compilation

- Remember to make sure you have all your files when you split them between .c and .h
- You include the .c files for compilation and the compiler will find the .h files.
- Object files unchanged.
Initializing the arrays are your problem

```c
int a[3];
....
X = a[1]; ....
```

Bound checking is your problem

```c
printf("%d",a[100]); ..... 
```

---

**Pointer power**

- Pointers are just 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 of pointers:
  ```c
  int *count;
  float *avg;
  ```
Pointers: nitty gritty

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

- example:
  
  ```
  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:

  ```
  int i = 3, j = -99;
  int count = 12;
  int *countPtr = &count;
  printf ( "%d", *countPtr);
  ```

- Here is the memory picture:
#include <stdio.h>
main() {
    int x, y;    // declare two ints
    int *px;    // declare a pointer to an int
    x = 3;      // initialize x
    px = &x;
    y = *px;
    printf( "x=%d px=%p y=%d\n", x, px, y );
}

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
Code

```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=0xbfffff4f0
i=0  arr[i]=29  &arr[i]=0xbfffff4f0
i=1  arr[i]=8   &arr[i]=0xbfffff4f4
i=2  arr[i]=18  &arr[i]=0xbfffff4f8
i=3  arr[i]=95  &arr[i]=0xbfffff4fc
i=4  arr[i]=48  &arr[i]=0xbfffff500
j=0xbfffff4f0  *j=29
after adding 1 to j:
  j=0xbfffff4f4  *j=8
Pointer operations

- 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

- storing multiple characters in a single variable
- data type is still char
- BUT it has a length
- 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
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 *s );
  ```
  this function returns the number of characters in s; 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

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

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

### 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
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....`
Malloc.c

#include <stdio.h>
#include <stdlib.h>
declare BLKSIZE 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 = BLKSIZE;
    buf = (char *)malloc( sizeof(char)*bufsiz );

    // read contents of file
    i = 0;
    while (( k = fgetc( fp )) != EOF ) {
        buf[i++] = k;
        if ( i == bufsiz ) {
            bufsiz += BLKSIZE;
            buf = (char *)realloc( buf,sizeof(char)*bufsiz );
        }
    }
    if ( i >= bufsiz-1 ) {
        bufsiz += BLKSIZE;
        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 );
  ```

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;  //MISTAKE THIS IS NOT A COPY

    ...
    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:
    ```c
    int vec[100];
    char str[30];
    float m[10][10];
    ```
  - dynamically:
    ```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 );
    ```
- for an array containing N elements, indeces 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! */
  ```
- 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
  - 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
- 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;
```

Code

```c
#include <stdio.h>
#define MAX 12
int main( void ) {
int x[MAX]; /* declare 12-element array */
int i, sum;
for ( i=0; i<MAX; i++ ) { x[i] = i; }
/* here, what is value of i? of x[i]? */
sum = 0;
for ( i=0; i<MAX; i++ ) { sum += x[i]; }
printf( "sum = %d\n",sum );
} /* end of main() */
```
Code 2

```c
#include <stdio.h>
#define MAX 10
int main( void ) {
  int x[MAX]; /* declare 10-element array */
  int i, sum, *p;
  p = &x[0];
  for ( i=0; i<MAX; i++ ) { *p = i + 1; p++; }
  p = &x[0];
  sum = 0;
  for ( i=0; i<MAX; i++ ) { sum += *p; p++; }
  printf( "sum = %d\n",sum );
} /* end of main() */
```

2 dimensional arrays

- 2-dimensional arrays
- 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 (which is an integer)!
swap

void swapNot( int a, int b ) {
    int tmp = a;
    a = b;
    b = tmp;
} // end swapNot()

void swap( int *a, int *b ) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
} // end swap()

int x, y; // declare two ints
int *px, *py; // declare two pointers to ints
x = 3; // initialize x
y = 5; // initialize y
printf( "before: x=%d y=%d\n", x, y );
swapNot( x, y );
printf( "after swapNot: x=%d y=%d\n", x, y );
px = &x; // set px to point to x (i.e., x's address)
py = &y; // set py to point to y (i.e., y's address)
printf( "the pointers: px=%p py=%p\n", px, py );
swap( px, py );
printf( "after swap with pointers: x=%d y=%d px=%p py=%p\n", x, y, px, py );
// you can also do this directly, without px and py:
swap( &x, &y );
printf( "after swap without pointers: x=%d y=%d\n", x, y );
Pointers

- Make sure you feel comfortable with the idea of what is happening inside pointer

- Will try to use more examples today to make specific points

```c
int main(){
    int number = 10;
    foo(&number);
    return 0;
}

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

- What is the advantage of passing in by pointer reference?
- What is the problem?
- How would we solve it?

const

- Allows the compiler to know which values shouldn’t be modified
- Added in to C later

Example:
```c
const int a = 5;

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