CS3157: Advanced Programming

Lecture #8
Oct 30
Shlomo Hershkop
shlomo@cs.columbia.edu
Announcements

- **Grades**
  - Apologize for grading delays
  - if you have concerns, please see me
    - if things are making sense, and you are submitting work, shouldn’t be concerned

- **Lab extension – 24 hours**

- **Homework**
  - Have homework 2 ready today

- **Please fill out the coursework survey**
  - will get credit in the course (I get an email who didn’t fill out the survey...rest will get credit)
Today

- more c
  - memory allocation
  - const
  - typedef
  - structs

- AI example
  - some games and strategies

- reading:
we were working on magic square generation

how did you find a magic square

any observations?
ideas

- what would have been a better way to go about it ??
Classic Example

- N queen Problem: how to place n queens on a chessboard without conflict
Any ideas

- how would you solve it in using random numbers?

- what is the overhead?

- here is an example of how to solve it
back to pointers

- lots of confusion in last week lab
  - strings
  - use of pointers
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 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
Example

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

- What is the difference ??

- Note: many people meant to do `strcpy(a1,"testing");`
Memory allocation

- when you have
  - int var1
    - compiler will set aside x bytes for storing int in var1
  - var1 = 19;
    - will place this value in memory set aside by var1
  - int ba[10];
    - will make 10 spots available, with first spot reachable through ba[0] to ba[9]
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:
- 0123456789abcdefghij

*(*(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];
const

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

- 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 same location, but the value can be modified

- `int * const ptr = &x;`

  `*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
  - `void foo (int , int (*)(int , int) );`
  
  ```c
  int example1(int x, int y) { return x+y; }
  
  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:

    ```
    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:**
  ```java
  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

  ```java
  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**
  ```java
  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
```c
#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?
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?
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:
  #include <stdlib.h>

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

- random number function returns a number between 0 and RAND_MAX (which is $2^{32}$)
  int i = rand();
```c
#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 );
}
```
Character handling

- character handling library
  
  ```c
  #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:
  ```c
  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)
  
  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

  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

- 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

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

- **case recognition**
  
  ```c
  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

  ```c
  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**
  
  ```c
  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)

  ```c
  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
ok we have covered the basics

what if you want to create new types?
Creating your own types

- Equivalent to a class idea in other programming languages, you can define your own types in C:

```c
struct name {
    types;
};
```
example

```c
struct point {
    int x;
    int y;
};

- **Usage:**
  ```
  struct point a;
  a.x = 5;
  a.y = 10;
  ```
Anonymous structs

- Can also create anonymous structs

```c
struct {
    int x;
    int y;
} a, b;
```
Nesting

struct rect {
    struct point pt1;
    struct point p2;
};

- Use:
  struct rect largeScreen;
Making space

- Remember in the proceeding examples, simple types so memory is automatically allocated (in a sense).
- struct student {
  char * name;
  int age;
};

struct student a;
a.name = (char*)malloc(sizeof(char)*25));
...


Use in functions

```c
struct point makePoint(int x, int y)
{
    struct point temp;
    temp.x = x;
    temp.y = y;
    return temp;
};
```
Operations

- Copy
- Assignments
- & (addressing)
- Accessing members

- How do we compare 2 structs
Structs and pointers

- struct point *example
  = (struct point *)malloc(sizeof(struct point));

- (*example).x

what does
*example.x mean?

Shortcut:
example->x
typedef

- defining your own types using typedef (for ease of use)
  typedef short int smallNumber;
  typedef unsigned char byte;
  typedef char String[100];

  smallNumber x;
  byte b;
  String name;
enum

- define new integer-like types as enumerated types:
  ```c
  enum weather { rain, snow=2, sun=4 };  
  typedef enum {Red, Orange, Yellow, Green, Blue, Violet} Color;
  ```

- look like C identifiers (names)
- are listed (enumerated) in definition
- treated like integers
  - start with 0 (unless you set value)
  - can add, subtract — e.g., color + weather
  - cannot print as symbol automatically (you have to write code to do the translation)
enum

- just fancy syntax for an ordered collection of integer constants:
  ```c
  typedef enum {
      Red, Orange, Yellow
  } Color;
  ```
- is like
  ```c
  #define Red 0
  #define Orange 1
  #define Yellow 2
  ```
- here's another way to define your own boolean:
  ```c
  typedef enum {False, True} boolean;
  ```
Usage

eenum Boolean {False, True};

...  
enum Boolean shouldWait = True;
...  
if(shouldWait == False) { .. }
struct

int main() {
    struct {
        int x;
        char y;
        float z;
    } rec;
    rec.x = 3;
    rec.y = 'a';
    rec.z = 3.1415;
    printf( "rec = %d %c %f\n", rec.x, rec.y, rec.z );
} // end of main()
```c
struct
int main() {
struct record {
    int x;
    char y;
    float z;
};
struct record rec;
    rec.x = 3;
    rec.y = 'a';
    rec.z = 3.1415;
printf( "rec = %d %c %f\n", rec.x, rec.y, rec.z );
} // end of main()
```
int main() {
    typedef struct {
        int x;
        char y;
        float z;
    } RECORD;
    RECORD rec;
    rec.x = 3;
    rec.y = 'a';
    rec.z = 3.1415;
    printf( "rec = %d %c %f\n", rec.x, rec.y, rec.z );
} // end of main()
note the use of malloc where "sizeof" takes the struct type as its argument (not the pointer!)

```c
int main() {
    typedef struct {
        int x;
        char y;
        float z;
    } RECORD;
    RECORD *rec = (RECORD *)malloc( sizeof( RECORD ));
    rec->x = 3;
    rec->y = 'a';
    rec->z = 3.1415;
    printf( "rec = %d %c %f\n", rec->x, rec->y, rec->z );
} // end of main()
```
Important to understand

- overall size of struct is the sum of the elements, plus padding for alignment (i.e., how many bytes are allocated)
- given previous examples: sizeof(rec) -> 12
- but, it depends on the size and order of content (e.g., ints need to be aligned on word boundaries, since size of char is 1 and size of int is 4):

  ```c
  struct {
      char x;
      int y;
      char z;
  } s1;
  /* x y z */
  /* |----|----|----| */
  /* sizeof s1 -> 12 */
  
  struct {
      char x, y;
      int z;
  } s2;
  /* xy z */
  /* |----|----| */
  /* sizeof s2 -> 8 */
  ```
Reminder

- pointers to structs are common — especially useful with functions (as arguments to functions or as function type)
- two notations for accessing elements: (*sp).field or sp->field
- (note: *sp.field doesn’t work)

```c
struct xyz {
  int x, y, z;
};
struct xyz s;
struct xyz *sp;
...
s.x = 1;
s.y = 2;
s.z = 3;
sp = &s;
(*sp).z = sp->x + sp->y;
```
Arrays of structs

- notations for accessing elements: `arr[i].field`

```c
struct xyz {
    int x, y, z;
};
struct xyz arr[2];
...
arr[0].x = 1;
arr[0].y = 2;
arr[0].z = 3;
arr[1].x = 4;
arr[1].y = 5;
arr[1].z = 6;
```
unions

- union
- like struct:
  ```c
  union u_tag {
    int ival;
    float fval;
    char *sval;
  } u;
  ```
- but only one of ival, fval and sval can be used in an instance of u (think container)
- overall size is largest of elements
Example

```c
#define NAME_LEN 40

struct person {
    char name[NAME_LEN+1];
    float height;
};

int main( void ) {
    struct person p;
    strcpy( p.name,"suzanne" );
    p.height = 60;
    printf( "name   = [%s]\n",p.name );
    printf( "height = %5.2f inches\n",p.height );
} // end of main()
```
Files

- so perl makes working with files a 3 line process

```perl
open (FH,"a.txt");
while(<>){
    chomp;
    print splice (split / / ) 1 1;
}
```
File Handling

- `File *log_file;`

- any ideas what this look like?
- use function fopen to open handle
- pass in arguments to fopen to set type
  - r  read
  - w  write
  - a  append
- need to check if not null
if( (log_file = fopen("some.txt", "w")) == NULL) {
    fprintf(stderr,"Cannot open %s\n", "log_file");
}

/*****
do your cool stuff here

*****/

cfclose(log_file);
moving characters

- can move characters using putchar(c) and getchar()
  - if no handle supplied
  - putchar(c, stdout)
  - getchar(stdin)
strings

- fgets
- fputs
Next lab

- Will be clearing next 2 labs to work on homework

- homework 2 released today
sudoku

- Have you heard of it??
- how is it played??
- think you can code it?
Homework

- don’t need to code main routines

- will need to use sudoku c code to run a web frontend to
  - get solutions
  - propose puzzles
  - analyze the c code