Reviewing some C features

1, Header files and function prototype

- Headers and include files, Prototype of functions are put in head files, including input type and output type.
- The C compiler works more dependably when it knows the input and output type of functions before it compiles a function call. Otherwise, it tries to guess, which is bad.
- Function prototypes solves this problem by explicitly describing the function, such as int foo(double x, int y).
- We usually put prototypes at the top of a c file or (better yet) in a separate head file, (.h) file.
- When we need to use functions from a .c file outside of that c file, we should just include the head file.

Example of Header files:

hellomake.c

```
#include “hellomake.h”
int main()
{
   // call a function in another file
   myPrintHelloMake();
    return(0);
 }
```

hellofunc.c

```
#include <stdio.h>

void myPrintHelloMake(void)
{
   printf("Hello makefiles!\n");
   return;
}
```

hellomake.h

```
/* example include file */
 void myPrintHelloMake(void);
```

makefile

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h

%.o: %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)
hellomake: hellomake.o  hellofunc.o
        gcc -o hellomake hellomake.o hellofunc.o -I.
```

More example;

Ex5.c

```
#include “cube.c” //
#include <stdio.h> //input and output, printing
#include <stdlib.h> //utility including maloc, free, NULL.
#include <math.h> //abs(), sin() , cos()
                        //”-lm” option required for gcc
#include <pthread.h> //thread library, need “-lpthread” in gcc
void main() {
}
```

to compile,

gcc Ex5.c timing.c –lm –lpthread –o Ex5

to run,

./Ex5

2 Structures and typedef

A struct is similar to an java object with no methods.

typedef can be used to define new types.

```
typedef struct {
   float *matrix;
   int startRow;
   int endRow;
}jobSpec; // jobSpec could be used to store the
            //information that each thread needs.
            //in this example, the matrix is shared by all threads,
            //but each thread is only responsible to process the rows
            //in the range of startRow and endRow.
```

3, Pointer variables

```
int x = 1, y = 2, arr[10] = {3, 4, 5, 6, 7};
int *p= &x;
*p = &y;
*p = 200;
*p = arr;
int **q = &p;  // what is the content in q?
```

•More examples

```
int *p = arr;
arr[3] = 14;
*(arr+3) = 14;
p[3] = 14;  //these three statements do the same thing.
```

4, Obtain memory on the heap

Question: have you ever come across segmentation fault (bus error, access violation) error in your own program? How do you address?

We can dynamically allocate memory from the heap using malloc().

For example,

```
float *fa = (float*) malloc(numberOfElement * sizeof(float));
if(fa == NULL)
{
   printf("Error, out of memory");
}
fa[12] = 47.3f;
```

After using fa, it is the programmer's responsibility to free it,

free(fa); //this cause trouble sometime in multi-thread programming,

//think why?

Note: malloc() returns generic pointer **void ***.

5, Passing Arguments

1)pass by value: passing a **copy** of the value contained in the argument. The function works with a copy, can not change the original.

2)pass by reference: passing the address of the argument variable,

could change the original.

3)thinking: how about in Java?

•Examples

```
int main() {
   int t = 0;
   sum(&t, 2, 4);
}
```

```
void sum(int *total, int x, int y){
   *total = x + y;
}//what is t after run the main() ? t is changed after call by reference.
```

The following is another way to return data from inside a C function.

```
float *readNewArray( int *rows, int *cols, const char *file) {
    //…..do work
    return fltPointer;
}
```

what is keyword “const”? and how to use this function in main?

```
void main(
   int rows, cols;
   float *arr = readNewArray(&rows, &cols, “foobar”);
}
```

The **const** mean read only, no modification,

const char *file = “fileName”; //mean the content that “file” pointer points to can not be modified.

How about this one? char * const file = “fileName”; ??

6, Command-line Arguments in C

```
int main( int argc, char * argv[]){
    if(argc != 3){
        usage();
        return 1;
    }
}
```

if we run a program on a terminal

% ./invert az100.in P

The content in argv are as following:

argv[0] → ./invert

argv[1] → az100.in

argv[2] → P

7, 2D matrix presentation

if we need a matrix with ncols = 4 and nrows = 3,

we can represent a 2D matrix using 1D array,

float *array = (float*)malloc(nrows * ncols * sizeof(float));

Logical 2D matrix layout,

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	
			(2,3)

If we store the matrix **physically in memory** in a row-major fashion, the matrix is stored **physically** in memory like this,

array →

(0,0)	(0,1)	(0,2)	(0,3)	(1,0)	(1,1)	…….

To reference a element (r, c) in 2D matrix,

array[r*ncols + c]

```
#define ROWS 100
#define COLS 200
void readData( void **data, int r,  int c);
typedef struct {
   float *matrix;
   int startRow;
   int endRow;
}jobSpec;

int main() {
   jobSpec *js = (jobSpec *) malloc( sizeof (jobSpec) );
   float *data = NULL;
   readData( (void **) &data, ROWS, COLS );
   js->matrix = data;
   // process data below
}

// Use call by reference to bring (*data2) outside of the function.
// That is, if we change data2 itself in readData, we cannot see the effect outside of the function.
// If we change what the pointer data2 points to, then we can see the effect outside of the function.
void readData( void **data2, int r,  int c) {
   int i, j;
   float *temp;
   *data2 = malloc(r * c * sizeof(float)); // why should we use *data here? why we should allocate memory first?
   temp =  (float *) (*data2);
   //after we initialized the memory space for data, we populate the data array
   for( i = 0; i < r; i ++ )
      for( j =0; j < c;  j ++ )
            temp[ i * c + j ] = readNextFloat();
}
```

Summary Today:

- C header files
- C pointers and double pointers
- Call by reference and value
- C structures and typedef
- Dynamic memory allocation
- C command line arguments
- 2D matrix represented with 1D array in C.

- Next Class
  - Threads and Pthreads Library
  - Classification of computer architectures
    - Flynn Taxonomy