Structure and Input/Output

Song Liu (song.liu@bristol.ac.uk)

GA 18, Fry Building,

Microsoft Teams (search "song liu").

Review

- So far, we have covered
  - Basics of Programming (Lec 1)
  - Functions (if-else statement) (Lec 2)
  - Arrays + Loops (Lec 3 + 4)
  - Pointers + Memory (Lec 5 + 6)
- This concludes Part I of C programming language which contains basic features of C.
- From today, we will progress to some more advanced features of C programming language.

Grouping

Grouping Variables

- We have learned how to declare **separate** variables:
  - `double pi = 3.141592654;`
  - `int studentID = 195235;`
- In many applications, variables go "hand-in-hand".
- To compute matrix multiplication of two matrices, we need **multiple variables** to describe each matrix involved in the computation:
  - a 2D array storing all the elements of matrix
  - number of rows
  - number of columns.

Revisit: Matrix Multiplication (Lec 4)

- This can lead to some complicated coding...
- We can do something like this:
- ```
  int num_col_A = 12, num_col_B = 13;
  int num_row_A = 13, num_row_B = 12;
  int elements_A[num_row_A][num_col_A];
  int elements_B[num_row_B][num_col_B];
  //... initialize elements_A and elements_B

  //call a function to
  multiply(num_row_A, num_colA, elements_A,
           num_row_B, num_colB, elements_B,
           elements_C);
  ```
- Lots of variables! Lots of repetition in names!!
  ```

Revisit: Matrix Multiplication (Lec 4)

- Suppose you made typo...
- ```
  int num_col_A = 12, num_col_B = 13;
  int num_row_A = 13, num_row_B = 12;
  int elements_A[num_row_A][num_col_A];
  int elements_B[num_row_B][num_col_B];
  //... initialize elements_A and elements_B

  //call a function to
  multiply(num_row_A, num_colB, elements_A,
           num_row_B, num_colB, elements_B,
           elements_C);
  ```
- Can you spot it?
- If you cannot spot it, your code will compile, run, but will NOT give you desired results!
  ```

Grouping Variables

- Since `num_col_A, num_row_A, elements_A` are all variables describing matrix A, can we group them together in a less complicated way?
- Introducing a C language feature: **Structure**.
- A structure groups several related variables into **a single entity.**

Structure

- Syntax for defining a structure:

```
struct structure_name{
    data_type variable1;
    data_type variable2;
    ...
};
```

  - Do not forget the `;` at the end!

- Syntax for declaring a structured variable

```
struct structure_name struct_variablename;
```

- Syntax for referencing a sub-level variable contained in a structure variable

```
struct_variablename.variable1
```

Example: Student

- First, let us study a toy example.
- Define a structure, `student` which contains three sub-level variables `ID`, `name` and `grade`.

Example: Student

Define a structure "student"

```
struct student{
    int ID;
    char *name;
    int grade;
};
```

Declare a structure variable and initialize it:

```
struct student song;
song.ID = 1024;
song.name = "song liu";
song.grade = 70;
```

print out `song` 's name:

```
printf("%s\n", song.name);
```

Example: Student

When initializing a structure variable, you can use a syntax that is similar to the array initialization:

```
struct student song = {1024, "song liu", 70};
printf("%s\n", song.name);
//displays: song liu
```

It only works when initializing! You **CANNOT** do

```
struct student song;
song = {1024, "song liu", 70};// COMPILATION ERROR!!!
```

Example: Student (full code)

```
#include <stdio.h>
struct student{
    int ID;
    char *name;
    int grade;
}; //Define a structure before you use it!!

void main(){
    struct student song; //declare a student variable
    //initialize
    song.ID = 1024;
    song.name = "song liu";
    song.grade = 70;

    printf("%s\n", song.name); //displays "song liu"

    //declare + initialize in one line.
    struct student song2 = {1024, "song liu", 70};

    printf("%s\n", song2.name); //displays "song liu"
}
```

Passing by Value

- Structures are **passed by value.**
  - This behavior is different from arrays, who are passed by reference!

```
#include<stdio.h>
struct student{
  int ID;
  char *name;
  int grade;
};

void hack(struct student s){
  s.grade = 9999;
}
void main(){
  struct student song = {1234, "song liu", 70};
  hack(song);
  printf("%d\n", song.grade);//display 70, not 9999!
}
```

Example: Matrix

- Define a matrix structure

```c
struct matrix{
   int numrow, numcol;
   int *elements; //pointing to flattened matrix
};
```

- Declare a `matrix` structure variable and initialize it

```c
struct matrix A;
A.numrow = 10;
A.numcol = 10;
// allocate heap memory for the matrix
A.elements = calloc(10*10, sizeof(int));
```

- Or in one line

```c
struct matrix A = {10,10,calloc(10*10, sizeof(int))};
```

Example: Matrix Multiplication 2.0

- Write a function computes the matrix multiplication, using `struct matrix` as inputs.

```
void multiply(struct matrix A,
              struct matrix B,
              struct matrix C){
//... implementation will be a part of lab.
}
```

- Finally, we can call the `multiply` like this:

```
multiply(A,B,C);
```

- The interface and usage of function `multiply` is much cleaner the earlier version.

Example: Matrix Multiplication 2.0

- You may be annoyed by the `struct` keywords appear at the definition of function `multiply`:

  ```
  void multiply(struct matrix A,
                struct matrix B,
                struct matrix C){
   //...
   }
  ```

- You can avoid the repetition of `struct` by adding

  ```
  typedef struct matrix Matrix;
  ```

  after the `struct matrix` definition.

  From now on, `Matrix` will be an alias of `struct matrix`.

Example: Matrix Multiplication 2.0

For example:

```
struct matrix{
    //... same as above
};
// add this line!
typedef struct matrix Matrix; //don't forget ";"!!
```

Then define your function using Matrix **without** `struct`.

```
void multiply(Matrix A, Matrix B, Matrix C){
    //...
}
```

Arrays in Structure

- You may wonder why don't we define a structure like this:

```
struct matrix{
    int numrow, numcol;
    int elements[numcol][numrow];
};// WRONG! COMPILATION ERROR!
```

- Although we can use arrays in structures, the sizes of arrays must be constants.

```
struct matrix{
    int numrow, numcol;
    int elements[4][4];
};// OK!
```

- However, it defeats the purpose of using a structure to store matrices with different sizes.

Input and Output (IO)

Overview

- In C, all IO operations are handled by function calls.
  - We have already encountered one such function
  - `printf(...)`
- Thanks to the abstraction of hardware, whatever IO devices you are using, these function calls are exactly the same.
- Today, the IO functions in C still inspire IO function designs in other programming languages.
- Here, we are going to focus on File IO.

Open a File `fopen`

- Usage: `FILE *fopen(char *filename, char *mode)`
  - `filename`: string, file name.
  - `mode`: access mode, can be
    - `"r"`: read-only, file must exist.
    - `"w"`: write, create an empty one if file does not exist.
    - `"r+"`: read and write, file must exist.
    - `"w+"`: read and write, create an empty file if file does not exist.
    - `"a"`: appending, create an empty one if file does not exist.
    - `"a+"`: appending and reading, create an empty one if file does not exist.

Open a File `fopen`

- Usage: `FILE *fopen(char *filename, char *mode)`
- `fopen` returns a pointer to a `FILE` structure.
  - You do not need to understand what `FILE` structure is. The definition of `FILE` structure is not visible to you.
  - This pointer is needed for further operations on the file.

Close a File `fclose`

- After read/write operations on a file, you MUST close it.
- Usage: `int flose(FILE * file)`
    - The input is the pointer you obtained from `fopen`.

Stream

- The design of C's IO functions are heavily influenced by the IO devices in the 60s, 70s.
    - These devices are mostly sequential and can move along one direction, such as tapes.
    - You can only read/write one byte after another.
    - Like a riding boat in a river...
- The abstraction of such devices is called IO Stream.
    - IO functions can only read or write "the next thing" in the stream.
    - The `FILE *` pointer indicates our current position in the stream.

Read the next Byte `fgetc`

- `int fgets(FILE *file)`
  - `file`: the pointer you obtained using `fopen`.
  - Returns the next byte in the stream, as an `int` variable.

Write the next Byte `fputc`

- `int fputc(int byte, FILE *file)`
  - `file`: the pointer you obtained using `fopen`.
  - `byte`: the byte to be written.
- When using `fgetc` or `fputc`, you need to set the `mode` in `fopen` to be `wb`, `rb` or `ab`, where `b` stands for binary.
  - This avoid extra handling on the line breaks.

Read the next Line `fgets`

- `char *fgets(char *line, int max, FILE* file)`
  - `line`: a pointer to an char array where the line is going to be stored.
  - `max`: the maximum number of character to be read.
  - `file`: the pointer you obtained using `fopen`.

Write the next Line `fprintf`

- `int fprintf(FILE *file, char *line, variables)`
  - `file` : the pointer you obtained using `fopen` .
  - `line` : the formatted string containing specifiers, like the one in `printf` .
  - `variable` : variables corresponds to the specifiers in `line` .
- `fprintf(file, "pi is %.2f.\n", 3.14)`
  - Write a line "pi is 3.14." to `file`

Example: Reading Lines from File

```c
#include <stdio.h>
void main()
{
    FILE *f = fopen("poem.txt", "r");
    char line[1024];
    while (1){ // loop forever until reach the end
        fgets(line, 1024, f); // read the next line
        if (feof(f)){
            break;// stop the loop if we are at
        // the end of the file
        }

        //print the line to screen
        printf("%s", line);
    }
    fclose(f);
}
```

Is this the end of file? `feof`

```c
while (1){
  // ...
    if (feof(f)){
        break;
    }
  //...
}
```

- As we read/write the next byte/line, we push the `FILE` pointer further down the IO stream until it reaches the End of File (EOF).
  - We can test whether EOF has been reached using the `FILE` pointer.
- `int feof(FILE *file)`
  - `file`: the pointer you obtained using `fopen`.
  - returns non-zero value if the we are at the end of the IO stream. Otherwise, return 0.

Conclusion

- Structure is a mechanism in C that groups related several variables together as a single entity.
  - Student example
  - Matrix example (contains a flattened matrix)
- Input and Output (IO)
  - File IO in C is handled as streams.
  - Read/Write a byte `fgetc`, `fputc`.
  - Read/Write a line `fgets`, `fprintf`.

Homework: Preparation

Open the skeleton file:

1. Create a matrix structure: containing three sub-level variables: `numrow`, `numcol`, `elements`.
2. Define `Matrix` as an alias of `struct matrix` using `typedef` statement.

Homework: Preparation

3. Write a function

```
int idx(int i, int j, Matrix m)
```

   - It takes the 2D index `i, j` of a matrix `m`, and converts it to the linearized index.
   - For example, if `m` is a 10 by 2 matrix, then
     - `idx(0, 0, m)` returns 0
     - `idx(0, 1, m)` returns 1
     - `idx(1, 0, m)` returns 2
     - `idx(1, 1, m)` returns 3
     - ...
   - The function `idx` should contain only one line of code.

Homework: Read

3. Write a function that reads a matrix from file.

```
Matrix read_matrix(char *filename)
```

   - `filename` is the name of a matrix file that contains a matrix A.

Homework: Read

This matrix file stores a flattened matrix

- The first 4 bytes contains an integer indicating how many rows does A have.
- The next 4 bytes contains an integer indicating how many columns does A have.
- The next byte contains an integer, which is A_{11}.
- The next byte contains an integer, which is A_{12}.
- ...
- The ncol + 1 byte contains an integer, which is A_{21}.
- The ncol + 2 byte contains an integer, which is A_{22}.
- ...

Homework: Write

4. Write a function that writes a matrix to file.

   ```
   void write_matrix(Matrix m, char *filename)
   ```

   - `m` is the matrix to be written.
   - `filename` is the name of a matrix file that will contain matrix `m`.
   - The format of this matrix file is the same as the one we described in the previous task.
5. Using the code in the skeleton file to test your implementation `read_matrix` and `write_matrix`.

Homework: Matrix Multiplication

6. Write a function that computes matrix multiplication:

   ```
   void multiply(Matrix A, Matrix B, Matrix C)
   ```

   It computes AB and assign the output to C.

Homework: Matrix Multiplication (submit)

6. Using the functions you have written above, complete the following task:
    i. Load matrix A from "A.matrix" and B from "B.matrix".
    ii. Compute $C = AB$.
    iii. Write matrix C to the file "C.matrix".

Homework: Matrix Multiplication (submit)

7. Modify your `multiply`, so it will check whether the input arguments `A`, `B` and `C` has the correct sizes.
    - If not, multiply will print out a message "check matrix sizes" and will exit without performing matrix multiplication.

Homework: Matrix Multiplication (challenge)

8. Using the functions you have written above, complete the following task:
   i. Load matrix A from "A.matrix" and B from "B.matrix".
   ii. Compute $C = A^\top B$, where $A^\top$ is the transpose of matrix A.
   iii. Write matrix C to the file "C.matrix".
   iv. Design your code so that it is efficient in terms of memory usage.
   v. You are not allowed to change the `multiply` function.