

Structure and Basic File IO

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

GA 18, Fry Building,

Microsoft Teams (search "song liu").

Previously

- Dynamic Memory (Heap Memory) Allocation
 - **Pros:** Can be allocated at **runtime**.
 - **Cons:** Must be released manually.
- `malloc` and `free`.
 - Allocate and release heap memory.
 - `malloc` returns a pointer pointing to the head of a continuous memory.
 - `free` frees the memory given a pointer.
- Usage of Heap Memory
 - Use pointer to access the variable.
 - **Use the array syntax if you want to access consecutive elements in the memory.**

Comparison with Stack Memory

1. Commonalities

- i. They both are parts of virtual memory.
- ii. They both can store variables/arrays.
- iii. When not initialized, the variables contain garbage values.

2. Differences:

- i. Stack Memory can only be allocated at the compilation time, by the compiler.
- ii. Heap Memory can only be allocated at the runtime, by the programmer.
- iii. Heap Memory needs to be managed by the programmer while Stack memory does not.

Row Major and Column Major

- Row Major and Column Major are two methods storing a matrix in an array.
- Matrix is a "2D object", you need to flatten it before storing it in a sequential container (such as an array).
- Use zero-based indexing (indices i, j starts from 0),
- Row-major order stores a matrix as

$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \end{bmatrix} \implies [A_{00}, A_{01}, A_{02}, A_{10}, A_{11}, A_{12}]$$

- Row major order means $A_{i,j}$ is the $i * \text{ncol} + j$ -th element in the array.

Row Major and Column Major

- Column-major order stores a matrix as

$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \end{bmatrix} \implies [A_{00}, A_{10}, A_{01}, A_{11}, A_{02}, A_{12}]$$

- Column Major order means $A_{i,j}$ is ?-th element in the array

Row Major and Column Major

- In the exam and future CWs, I will say something like:
 - "an array `a` stores a matrix $A \in \mathbb{N}^{m \times n}$, in row-major order. "
 - You should know what I mean by that!

Today

1. Structure: Group variables together.
2. File IO functions: Operating on files.

Grouping



Grouping Variables

- We have learned how to declare **individual** variables:
 - `double pi = 3.141592654;`
 - `int studentID = 195235;`
- In many applications, variables are bundled together and should be dealt with as a group
- For example:
 - `studentID` are commonly associated with other basic information, such as `name`, `age`, `grade` etc.
 - An administrative software would like to handle these variables as a group of variables, rather than individual variables.

Example: Vector Operations

- Print vector

```
void print(double vec1[], int len1){
    for(int i = 0; i<len1; i++){
        printf("%.2f ", vec1[i]);
    }
    printf("\n");
}
```

- Vector dot product

```
double dot(double vec1[], int len1,
            double vec2[], int len2){
    // check if len_vec1 == lev_vec2 ...
    double sum = 0;
    for(int i = 0; i<len1; i++){
        sum += vec1[i]*vec2[i];
    }
    return sum;
}
```

Example: Vector Operations

- Declare and initialize vectors (arrays)

```
int len1 = 3; int len2 = 3;  
double *vec1 = malloc(len1*sizeof(double)); //heap mem  
double *vec2 = malloc(len2*sizeof(double)); //heap mem
```

- Call vector functions

```
print(vec1, len1);  
print(vec2, len2);  
double d = dot(vec1, len1, vec2, len2);  
// free heap memory, don't forget!
```

- The array and its length are always **linked** in our code.
- Having to write the array and its length in every function seems unnecessarily complicated!

Example: Vector Operations

- Would be nice to call functions like this:

```
//v1 and v2 contains the information about its length  
print(v1);  
double d = dot(v1, v2);
```

- Python programs work in this way:

```
>>> v1 = [1, 2, 3]  
>>> print(v1)  
[1, 2, 3]
```

- How can we do it in C?

Grouping Variables

- Since `len1, vec1` are all variables describing the vector, we can group them together.
- 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; //trying to hack the score!
}

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

Example: Vector

- Define a `vector` structure

```
struct vector{  
    int len;  
    int *elements; //pointing to the array  
                    //stores elements of the vector.  
};
```

- Declare a `vector` structure variable and initialize it

```
struct vector v;  
v.len = 10;  
// allocate heap memory for the vector  
v.elements = calloc(v.len, sizeof(int));
```

- Or in one line

```
struct vector v = {10, calloc(10, sizeof(int))};
```

Example: Vector Operations 2.0

- Write a function that prints a vector, using `struct vector` as the input.

```
void print(struct vector v){  
    // v.len contains the length of the vector!  
    for (int i=0; i<v.len; i++){  
        printf("%d ", v.elements[i]);  
    }  
    printf("\n");  
}
```

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

```
print(v);
```

- The interface and usage of function `print` is much cleaner the earlier version.
- Implementing other vector operations using `struct` will be part of your lab this week.

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 fclose(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 fgetc(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.

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 formatted string `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`, like in `printf`.
- `fprintf(file, "pi is %.2f.\n", 3.14)`
 - Write a line "pi is 3.14." to `file`

Example: Reading Lines from File

```
#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`

```
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
 - Vector example
- Input and Output (IO)
 - File IO in C is handled as IO streams.
 - Read/Write a byte `fgetc` , `fputc` .
 - Read/Write a line `fgets` , `fprintf` .

Lab 1

1. Download the lab file and unzip it to your lab pack folder.
2. Read `student.c` and see how a structure is defined and sub-level variables are initialized and accessed.
3. Define a `print` function takes a `struct student` variable and print out student's information in a easy to read manner. For example:
 - `ID: ..., Name: ..., Grade: ...`

Lab 2

4. Define a `struct student` array with three elements.
 - You can populate the array with dummy student information.
5. Write a `pass_check` function that takes an array of `struct student` variables and print the info of students who has `grade>40`.
 - Print one student per line.

Lab 3

1. Open `vector2.c` and see how the `vector` structure is defined.
2. Write the `dot` function that takes two vectors u, v and compute their dot product $u \cdot v$.
3. Write the `norm` function that computes the norm of a vector v .

$$\text{norm}(v) = \sqrt{v \cdot v}$$

4. Write the `dist` function that computes the euclidean distance between its input u and v :

$$\text{dist}(u, v) = \sqrt{u \cdot u + v \cdot v - 2u \cdot v}.$$

5. Test your functions in `main` with provided test cases.

Lab 4 (submit)

Now, let us adapt the vector structure and create a matrix structure.

1. Create `ab1234.c`, where `ab1234` is your email before @.
2. Create a structure called `matrix`, containing three sub-level variables: `m`, `n`, `elements`, where
 - i. `elements` is an integer pointer, pointing to an array which stores a matrix $A \in \mathbb{N}^{m \times n}$ in row-major order.
 - ii. `m` and `n` are integer variables.
3. Create a `matrix` structure variable `m1`, and initialize it with a 2 by 3 matrix:

$$A = \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \end{bmatrix}.$$

4. Write a function `get_elem`, which takes three inputs `m, i, j`. i, j are **zero-based indices** and `m` is a matrix structure. It returns the i, j -th entry of A stored in `m`.
5. Test your function using `m1` you have just created.