CSCI 3150 Lab2

Caius (Zijie Dai) 1155141656@link.cuhk.edu.hk

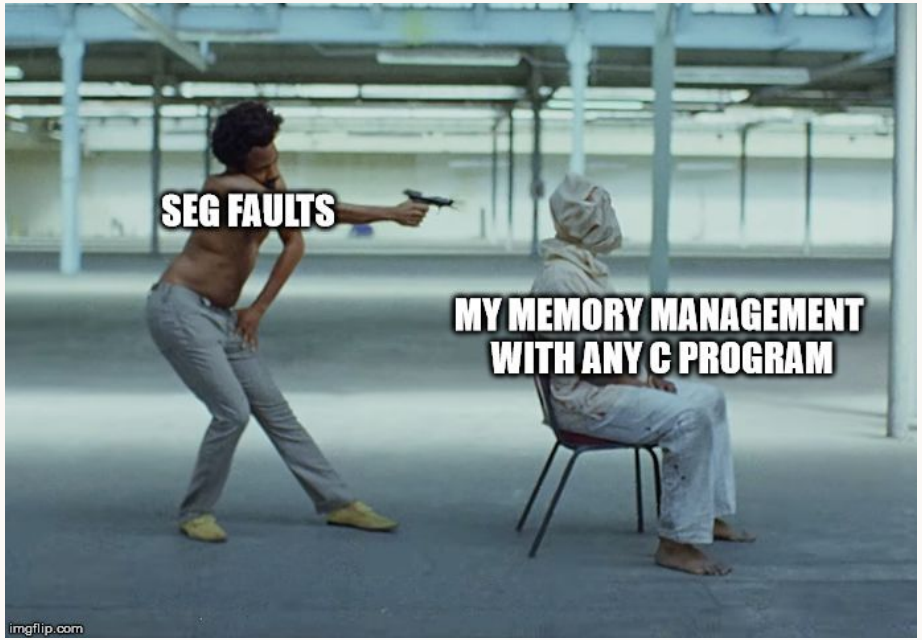
Lab 2 Contents

- Mainly focus on C programming review - **memory management** , **file I/O** and **structure**,. These concepts are essential for you to finish later labs/ass/proj.

- Release of Lab Exercise 2. **(Due on Sept. 25th)**

Memory Management

- Stack vs Heap
- malloc() & free()
- Common pitfalls



Stack Memory

Stack Memory: Static size, managed by the compiler automatically. E.g.: local variables, function parameters…

```
void function() {
    int x = 10;          // Stack memory
    char buffer[100];    // Stack memory
}  // Memory automatically freed here
```

Stack objects' memory size must be known at compile time, meaning that if you want to create an array whose size can't be determined before starting your program, you can't use stack memory.

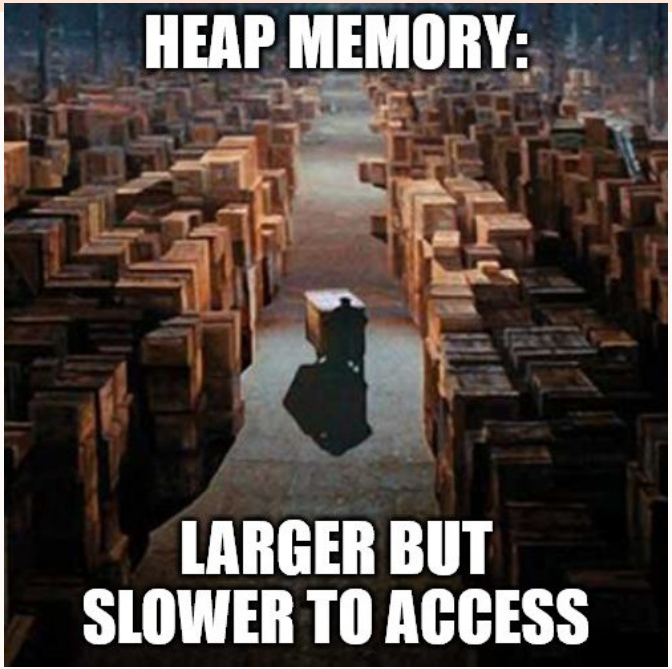
Stack Memory

- Stack memory requires **compile-time known sizes**
- Array sizes must be constant expressions
- What if we don't know the size until runtime?
- What if we need HUGE arrays that won't fit on the stack? (stack memory is usually much smaller comparing with heap memory)

```c
#include <stdio.h>

int main() {
    printf("How many numbers do you want to store? ");
    int num_entries;
    scanf("%d", &num_entries);
    int my_static_arry[100]; // Ok! Compiler knows the size
    int my_dyn_arr[num_entries];  // High chance to trigger error
    // "array size must be constant"
}
```

Heap Memory



- Dynamic size determined at runtime
- Much larger memory space (GB)
- Flexible lifetime management (**Managed by you**)
- Needs explicit control over creation and deletion.

Memory Allocation: void *malloc(size_t num_bytes) This function allocates an array of num_bytes and leaves them uninitialized. Returns a pointer

Memory Deletion: void free(void* address); This function releases a block of memory block specified by address.

To use them, include the <stdlib.h> header file.

Heap Memory

The program we show here computes an array with the following format:

0,2,4,6,8,10.....

It allows us to dynamically decide the array size. (note that the parameter for malloc is not num_entry, but num_bytes, thus we use num_entry * sizeof(entry_type) to decide the size.)

More examples:

```c
int *numbers = malloc(count * sizeof(int));
char *string = malloc((length + 1) * sizeof(char));
double *matrix = malloc(rows * cols * sizeof(double));
```

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    printf("How many numbers? ");
    int num_entries;
    scanf("%d", &num_entries);

    // Now we CAN create dynamic arrays!
    int *my_arr = malloc(num_entries * sizeof(int));

    if (my_arr == NULL) {
        printf("Not enough memory!\n");
        return 1;
    }

    // Use it like a normal array
    for (int i = 0; i < num_entries; i++) {
        my_arr[i] = i * 2;
        printf("%d ", my_arr[i]);
    }

    // Don't forget to clean up!
    free(my_arr);
    return 0;
}
```

Common Pitfalls

- Forgetting to free() → memory leaks
- Using array after free() → undefined behavior
- Freeing same memory twice → crash

Try to enable -Wall flag for gcc compiler which might give you a warning when you do the dangerous things

File I/O

What we'll cover: `open()`, `read()`, `write()`, `lseek()`, and file flags.

Typical workflow:
Step 1: Open a file, get a file descriptor (Conceptually: get file variable)
Step 2: Perform read or write

Open a file:

int open(const char* path,int flags);
E.g.: Open a file with path path/to/file, with creation and read-write access mode. Note: If you want to create a new file (by specifying **O_CREAT**), you need to input the third argument, specifying the access privilege for the new file. In most cases, input **S_IRUSR | S_IWUSR** is sufficient (for more details, check here) Otherwise the file access privilege may prevent you from opening them.

```
int file = open("path/to/file",O_CREAT|O_RDWR, S_IRUSR | S_IWUSR )
```

```
int file = open("path/to/file",O_RDWR);
```

O_RDONLY : Read only.
O_WRONLY : Write only.
O_RDWR : Read and write.
O_APPEND: The file is opened in append mode and the file offset is positioned at the end of the file.
O_CREAT: If the pathname does not exist, create it as a regular file.
O_TRUNC: If the file already exists and is a regular file and the access mode allows writing (i.e., is O_RDWR or O_WRONLY) it will be truncated to length 0.

File I/O - read() and write()

ssize_t read(int *fd*, void *buf*[*count*], size_t *count*);

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

Step 1: Open a file
Step 2: Declare a buffer array to store the data
Step 3: Read the data from file

```c
int fd;
fd = open("input_test.txt", O_RDWR);
if (fd < 0) {
    perror("Error opening file");
    return 1;
}
char read_buffer[20];
ssize_t bytes_read = read(fd, read_buffer, 14);
if (bytes_read < 0) {
    perror("Error reading from file");
    close(fd);
    return 1;
}else{
    printf("read %ld bytes to file\n", bytes_read);
}
close(fd);
```

ssize_t write(int *fd*, void *buf*[*count*], size_t *count*);

write() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

Tips: When writing a string into a file, you can use strlen() function.

```c
int wfd;
wfd = open("input_test.txt", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR );
if (wfd < 0) {
    perror("Error opening file");
    return 1;
}
const char *write_buffer = "Hello, World!\n";
ssize_t bytes_written = write(wfd, write_buffer, strlen(write_buffer));
if (bytes_written < 0) {
    perror("Error writing to file");
    close(wfd);
    return 1;
}else{
    printf("wrote %ld bytes to file\n", bytes_written);
}
close(wfd);
```

Filo I/O - lseek()

When we open a file with open(), internally, OS helps us to remember what's our current read/write position. Thus if we call read() to read 10 bytes and then call write() to write 10 bytes, we are actually writing from position 10-20. lseek() helps us to move this internal "pointer" to a desired place.

```
off_t lseek(int fd, off_t offset,
int whence);
```

lseek() repositions the file offset of the open file description associated with the file descriptor *fd* to the argument *offset* according to the directive *whence* as follows:

SEEK_SET
The file offset is set to *offset* bytes.

SEEK_CUR
The file offset is set to its current location plus *offset* bytes.

SEEK_END
The file offset is set to the size of the file plus *offset* bytes.

```c
char buf[20]="Hello World!";
char rbuf1[20], rbuf2[20];
int fd;

if( (fd = open("myfile", O_CREAT | O_TRUNC | O_RDWR, S_IRUSR | S_IWUSR )) < 0)
    printf("Error in open()\n");
    exit(-1);
}
printf("fd is %d\n", fd);
// Write 20 bytes
if( write(fd, buf, 20) < 0){
    printf("Error in write()\n");
    exit(-1);
}
// Move internal pointer to the beginning
if( lseek(fd, 0, SEEK_SET) < 0){
    printf("Error in lseek()\n");
    exit(-1);
}
// Read 20 bytes
if( read(fd, rbuf1, 20) < 0){
    printf("Error in read()\n");
    exit(-1);
}
printf("rbuf1: %s\n", rbuf1);

// Go to the 6th bytes
if( lseek(fd, 6, SEEK_SET) < 0){
    printf("Error in lseek()\n");
    exit(-1);
}

if( read(fd, rbuf2, 20) < 0){
    printf("Error in read()\n");
    exit(-1);
}
printf("rbuf2: %s\n", rbuf2);
close(fd);
return 0;
```

Struct

Motivation: we want to use higher level abstraction to represent some concepts beyond numbers and characters. For example, a book.

```
struct Book {
  char title[50];
  char author[50];
  char subject[100];
  int book_id;
}; // Note the semi-colon
```

```
//Syntax:
struct <Name> {
  // Fields
}; // Note the semi-colon
```

How to define struct and use it

```
struct Book {
  char title[50];
  char author[50];
  char subject[100];
  int book_id;
};

struct Book bookA;
bookA.book_id = 1;
bookA.author[0] = "A"
strcpy(bookA.title,"Lab02")
```

```
typedef struct Book {
  char title[50];
  char author[50];
  char subject[100];
  int book_id;
}MyBook;

MyBook bookA;
bookA.book_id = 1;
bookA.author[0] = "A"
strcpy(bookA.title,"Lab02")
```

```
typedef struct {
  char title[50];
  char author[50];
  char subject[100];
  int book_id;
}Book;

Book bookA;
bookA.book_id = 1;
bookA.author[0] = "A"
strcpy(bookA.title,"Lab02")
```

Pointers to structures

```
typedef struct {
  char title[50];
  char author[50];
  char subject[100];
  int book_id;
}Book;

void change_id(Book book, int id){
  book.book_id = id;
  return;
}

void change_id_2(Book* book,int id){
  book->book_id = id;
  return;
}

Book bookA,bookB;
change_id(bookA,2);
change_id_2(&bookB,3);
```

When pass the struct object into the function, usually we use the pointer type, otherwise a complete copy will be made, and any modification within that function will only happen within that function.

change_id() will create a complete copy of bookA, and operate on that copy. (bookA's id remains unchanged)

change_id_2() will not create a copy of bookB, the modification within it will take effect.

Lab Exercise (Due on Sept. 25th)

In the following exercise, you will need to read one file: input.txt, which contains 11 lines.

- First line contains the **base number** (b). You can safely assume b is a positive number and is strictly smaller than 1000.
- The remaining 10 lines each contains a **positive integer** number which **we guarantee is strictly smaller than 1000.**
- You need to read in each integer, add base number to it, and write the results line by line to another file called output.txt. (i.e.: output.txt should contain 10 lines, each line is the result from the addition)
- **Please only submit the exercise.c**
- **You can certainly use other C libraries for file I/O, as long as you can meet the requirement.**
- **DO NOT modify Makefile, as we will use the original Makefile during testing.**

Tips:
1. The newline character in Linux is '\n'
2. sscanf: You can use this function to convert char array to int (reference: http://www.cplusplus.com/reference/cstdio/sscanf/).
3. sprintf: You can use this function to convert int to char array (reference: http://www.cplusplus.com/reference/cstdio/sprintf/).

Further Notes

1. Do not hardcode the solution by peeking the provided input.txt. We will use a different input.txt for testing. (Same format, different numbers, all assumption still holds)

2. In lab02 folder, there are two sample files, sample_input.txt and sample_output.txt, you can check the expected result for references. (Or use it for your own testing purpose)

3. We will use the required environment (either docker or virtual machine) for the testing. Please use the required environment as in lab1. If your program can not compile, you might get 0 for this lab.