VG101: Introduction to Computer and Programming

Week9 Checklist

Debug

- breakpoint (demo)
- `clang` sanitize: `clang -O2 -Wall -pedantic -Werror -Wno-unused-result -std=c11 -lm -fsanitize=address -fno-omit-frame-pointer -fsanitize=undefined -fsanitize=integer -o main main.c`
- Example: when you miss a `\0`

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

int main()
{
    char hey[4] = "Hey!";
    char something_else[30] = " You shouldn't see this!\n";
    printf("%s", hey);
    printf("%p\n", (void*)hey);
    printf("%p\n", (void*)something_else);
    return 0;
}
/*
Output:
Hey! You shouldn't see this!
0x7ffee97e27bc
0x7ffee97e27c0
 */
```

Note that, the example above is actually undefined behavior, so it may vary in different compiler and different computer architecture, and is not guaranteed to happen exactly as above.

Further Tips about C-style String

- Difference between a `char` array and a C-style string
    - a `char` array is not necessary terminated by `\0` (null). It just represents an array, where each element is a char.

    - a C-style string must be terminated by `\0` (null). Almost all the functions in `<string.h>` is designed for string, so it will require `\0`. Becareful when you see a function requires a `char*` as a parameter. Think twice whether it is requred a string or simply a pointer to char.

```
char alphabet[26];
for (int i = 0; i < 26; i++)
    alphabet[i] = 'a' + i;          // Here alphabet is simply a
collection of char
```

- Declaration

```
char city[7] = "Dallas";    // Declare memory (with the size of 7 chars)
char *pcity = "Dallas";     // Don't write anything like this
```

- `int scanf(const char *restrict format, ...)`

> **DESCRIPTION**

> ... White space (such as blanks, tabs, or newlines) in the format string match any amount of white space, including none, in the input. Everything else matches only itself. Scanning stops when an input character does not match such a format character. Scanning also stops when an input conversion cannot be made (see below).

> **CONVERSIONS**

> s: Matches a sequence of non-white-space characters; the next pointer must be a pointer to char, and the array must be large enough to accept all the sequence and the terminating NUL character. The input string stops at white space or at the maximum field width, whichever occurs first.

> c: Matches a sequence of width count characters (default 1); the next pointer must be a pointer to char, and there must be enough room for all the characters (no terminating NUL is added). The usual skip of leading white space is suppressed. To skip white space first, use an explicit space in the format.

- `char * fgets(char * restrict str, int size, FILE * restrict stream)`: you can specify the source of reading (`stdin` or file)

> **DESCRIPTION**

> The `fgets()` function reads at most one less than the number of characters specified by size from the given stream and stores them in the string str. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. If any characters are read and there is no error, a `\0` character is appended to end the string.

```
char str[20] = {0};
fgets(str, 20, stdin);      // why it is considered safer?
FILE* f = fopen("hello.txt", "r");
fgets(str, 20, f);
fclose(f);
```

- `char * gets(char *str)`: it has been deprecated due to the safety concern (buffer overflow attack)

> **DESCRIPTION**
>
> The `gets()` function is equivalent to fgets() with an infinite size and a stream of stdin, except that the newline character (if any) is not stored in the string. It is the caller's responsibility to ensure that the input line, if any, is sufficiently short to fit in the string.

```
char str[20] = {0};
gets(str);
```

- `size_t strlen(const char *s)`: will not count the `\0`

```
char str[20] = "Hello";
int len = (int)strlen(str);     // len = 5
```

- `char * stpcpy(char * dst, const char * src)`: copy the string src to dst (including the terminating `\0` character.)

```
char str_src[10] = "Hello";
char str_dst[6];
strcpy(str_dst, str_src);
printf("%s\n", str_dst);    // Hello
```

- `char * strcat(char *restrict s1, const char *restrict s2)`

> **DESCRIPTION**
>
> The `strcat()` functions appends a copy of the null-terminated string `s2` to the end of the null-terminated string `s1`, then add a terminating `\0`. The string `s1` must have sufficient space to hold the result.

```
char str_1[20] = "Hello ";
char str_2[20] = "World!";
strcat(str_1, str_2);
printf("%s\n", str_1);      // Hello World!
```

- `int strcmp(const char *s1, const char *s2)`: do not use `==` to compare string! `==` is only comparing two pointer (two address).

> **DESCRIPTION**

> The `strcmp()` function lexicographically compares the null-terminated strings `s1` and `s2`.

> **RETURN VALUES**

> The `strcmp()` function returns an integer greater than, equal to, or less than 0, according as the string `s1` is greater than, equal to, or less than the string `s2`. The comparison is done using unsigned characters, so that `\200` is greater than `\0`.

- `char * strchr(const char *s, int c)`

> **DESCRIPTION**

> The `strchr()` function locates the first occurrence of `c` (converted to a char) in the string pointed to by `s`. The terminating null character is considered to be part of the string; therefore if c is `\0`, the functions locate the terminating `\0'.

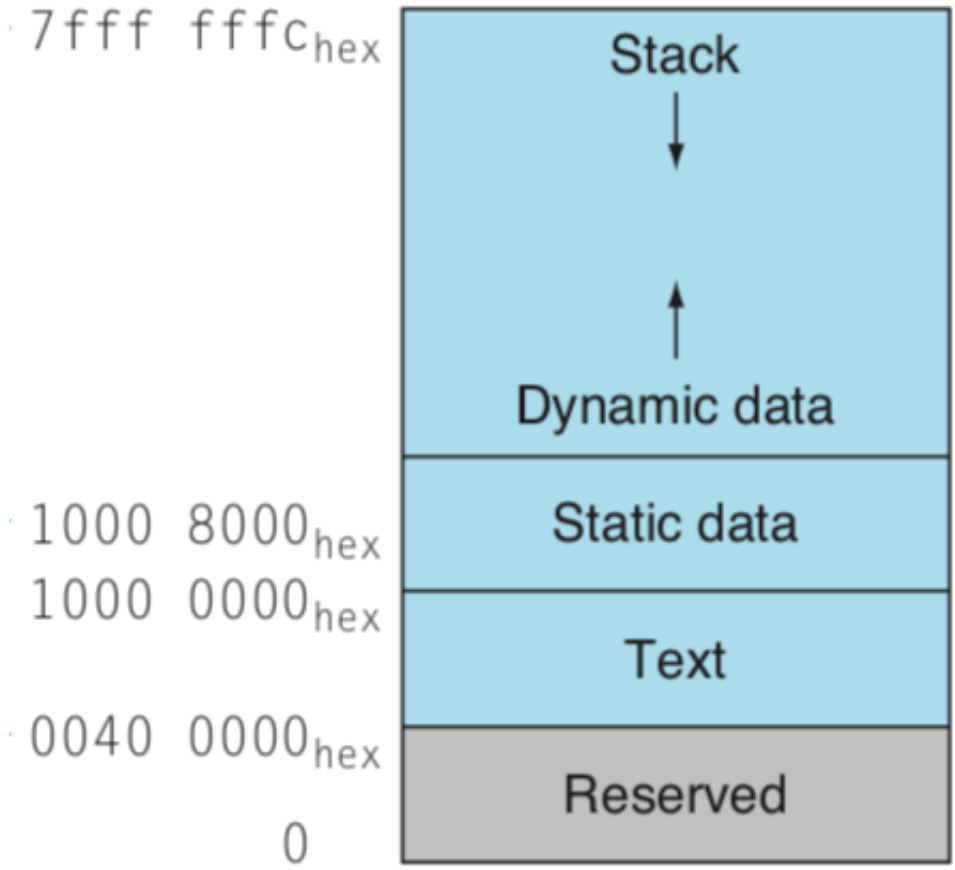
- `char * strstr(const char *haystack, const char *needle)`

> **DESCRIPTION**

> The `strstr()` function locates the first occurrence of the null-terminated string `needle` in the null-terminated string haystack.

- Convert string to numbers
  - `int atoi ( const char * str )`: Convert string to integer
  - `double atof ( const char * str )`: Convert string to double
  - `long atol ( const char * str )`: Convert string to long integer

Dynamic Memory Allocation



Stack

- Local variables are allocated in the stack
- Once the block execution is finished, the stack pointer will be recovered to the status before the block execution, and the local variables will be destroyed
- The size of the stack is limited, so do not declare a huge array as a local variable, which may result in "stack overflow"
- In short, the memory in stack is assigned to variables when the variable is declared, and freed when the variable is destoried (out of scope).

Heap

- The heap is a large pool of memory used for dynamic allocation. So when declare a huge array, we may prefer to put it in heap
- The memory dynamically allocated in the heap must be freed manually
- In short, the memory in heap is assigned and freed by the programmer manually. It means the programmer could control the life cycle of a variable, rather then collected by operatring system automatically. It can always be dangerous if the programmer doesn't handle it properly

`malloc()` and `free()`

- How could we declare memory in heap rather than in stack? Use `malloc()` and `free()`
- `#include <stdlib.h>`
- `void * malloc(size_t size)` : it will allocate memory from heap (the size of memory is

specified by the parameter `size`), and return the pointer to the first byte
- `void free(void *ptr)`: it will free the memory pointed by `ptr` in the memory. Question: how the program knows how much memory to free?
- Return `NULL` if fail to allocate memory.

```
int *arr = (int *) malloc(10*sizeof(int));      // allocate an piece of
memory
                                                // with the size of 10
ints
free(arr)                                       // free the memory
```

- Life cycles of allocated memory
- What's the point to allocate memory and free memory manually?
- You **must** free all the memory you allocate
- Memory leak
- Use `clang` sanitize to check memory leak

File I/O

- File pointer `FILE *`
- `fopen` and `fclose`
- Modes for `fopen`

Mode	Explanation
'r'	Open file for reading. The file must exist.
'w'	Open or create new file for writing. Discard existing contents, if any.
'a'	Open or create new file for writing. Append data to the end of the file.
'r+'	Open file for reading and writing. The file must exist.
'w+'	Open or create new file for reading and writing. Discard existing contents, if any. You can only read what you write.
'a+'	Open or create new file for reading and writing. Append data to the end of the file. You can read what already exists.

```
#include <stdio.h>
int main ()
{
    FILE * pFile;
    int n;
    char name [100];
    pFile = fopen ("myfile.txt","w");
    for (n=0 ; n<3 ; n++)
    {
```

```
        puts ("please, enter a name: ");
        gets (name);
        fprintf (pFile, "Name %d [%-10.10s]\n",n,name);
    }
    fclose (pFile);
    return 0;
}
```

- As always, you should close the file you open.

Structure

- In C, you can assemble a package of variables into a structure.
- Define your own datatype.
- `typedef` :use the syntax to declare an instance as the alias name for the type

```
typedef unsigned long ulong;
typedef int* ptr_int;
```

- Two ways to define strctures (they are equivalent):

```
struct Student
{
    char name[100];
    int studentID[12];
    int midtermScore;
    int finalScore;
};                      // need ";" at the end. Don't forget.
```

```
struct                  // An anonymous structure
{
    char name[100];
    int studentID[12];
    int midtermScore;
    int finalScore;
} a_student;            // Here it declare an instance of this anonymous
structure

typedef struct          // typedef: make an alias name for structure
{
    char name[100];
    int studentID[12];
    int midtermScore;
    int finalScore;
} student;              // Here it defines the structure as "student"
```

- Why structure? Make data easier to organize.

- How to use structure after defining it? Treat it as a new variable type.

```
struct Student
{
    char name[100];
    int studentID[12];
    int midtermScore;
    int finalScore;
};          // Usually written at the head of your program or in .h file

int main()
{
    struct Student MrBean;
    struct Student vg101[120];
    struct Student * JI = (struct Student *) malloc(300*sizeof(struct
Student));
    free(JI);
    return 0;
}
```

- To initialize structure:
- To access variable in structrue:

```
struct Student MrBean = {       // the standard way of initialization
    .name= "MrBean",            // or you can initialize member data one
by one by
    .studentID={0},             // MrBean.name = "MrBean";
    .midtermScore = 0,          // MrBean.studentID = {0};
    .finalScore = 0             // ...
};
struct Student MrsBean = {"MrsBean", {0}, 0, 0};
struct Student *pt = &MrBean;
printf("%s's midterm score = %d\n", MrBean.name, MrBean.midtermScore);
// use . to visit member data.
printf("%s's final score = %d\n", pt->name, pt->finalScore);
// use -> if using pointer.
```