C Programming Refresher

ENGG5105/CSCI5470
Tutorial 1

Jeremy Chan

Some notes from the CSE summer course of Dr. Matthew TANG & Dr. T.Y. WONG

http://appsrv.cse.cuhk.edu.hk/~csesc/

Logistics

- Assignment 1 is posted
  - **DUE**: Feb 13
  - **DEMO**: Feb 14
  - The lecture on Jan 22 will be very useful
- Tutorials
  - **Today**: C Programming
  - **Jan 23**: VM Usage + Assignment 1 Tips
  - **Jan 30**: Chinese New Year Holiday
  - **Feb 6**: Q & A / Office Hour (TBC)
  - **Feb 13**: No Tutorial (work on your assignment)

C Programming Refresher

- Helps you refresh your C programming skills in 45 minutes
  - Basics - e.g., strings, pointers, reading man pages
  - Pitfalls - e.g., strcpy vs memcpy
- Basic debugging skills that will be useful for your assignment
  - gdb
  - valgrind

Important Reminder

- Do all your programming work on Linux
  - The official testing environment is the Department VM

- No marks will be given if your program doesn't work in our Ubuntu, even if it runs perfectly on Mac / Windows

Data Type

- Char: `char (single quote) 'a' 'b' 'c'`
- Integer: `short, int, long, long long`
- Floating Point: `float, double`
- Boolean: `int 0 = false, other int = true`
- Class Object: No such things
- Array: Not an object
- String: Array of `char (double quote) "abc\0"`

Range of Data Types

- Pay great attention to range of data types
  - The following code produces no errors / warnings even with `-Wall`

```c
#include <stdio.h>

int main() {
    int a = 300;
    char c = a;
    printf ("%d\n", c);
    return 0;
}
```

String

```
char hello_string[100] = "hello world";
```

index	0	1	2	3	4	5	6	7	8	9	10	11
value	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	'\0'

- `int strlen(char str[])`
  - return length of the string, excluding '\0'
- `int strcmp(char str1[], char str2[])`
  - return 0 means str1 and str2 are identical
  - return 1 / -1 if str1 is lexicographically bigger / smaller than str2
- `char* strcpy(char dest[], char src[])`
  - copy src to the dest, stops at \0

strncmp ( ) vs memcmp ( )

index	0	1	2	3	4	5	6	7	8
str1	'h'	'e'	'l'	'l'	'o'	'\0'	'a'	'b'	'c'

index	0	1	2	3	4	5	6	7	8
str2	'h'	'e'	'l'	'l'	'o'	'\0'	'd'	'e'	'f'

```
strncmp (str1, str2, 9)    0
memcmp (str1, str2, 9)     <0
strncmp (str1, str2, 10)0
memcmp (str1, str2, 10)    undefined
```

Rule of thumb

Use memcmp() for binary data, strcmp() / strncmp() for strings

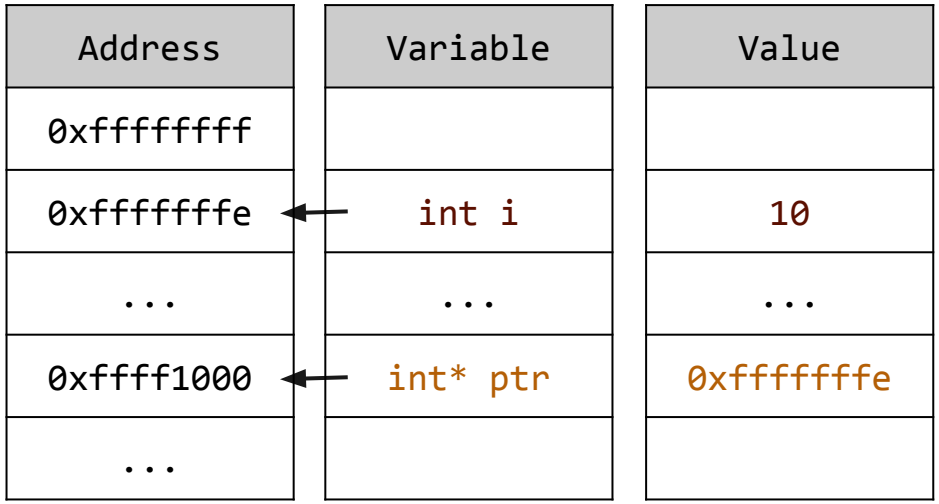
Pointer and Address

Address	Variable	Value
0xffffffff		
0xfffffffe	int i	10
...	...	...
0xffff1000	int* ptr	0xfffffffe
...		

Everything in C has an address in memory

```
int i = 10;
int* ptr = &i;
```

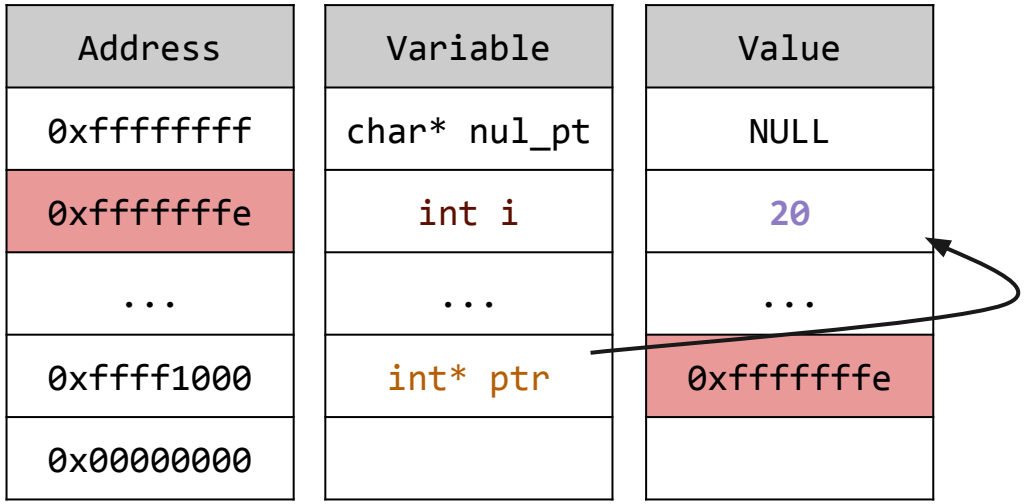
Pointer and Address



"&" means obtain the address

- `&i` equals `0xfffffffe`
- `&ptr` equals `0xffff1000`
- `ptr` equals `0xfffffffe`

Pointer and Address



"*" means access the value in the address

- `*ptr = 20;`
- `i` equals `20` from now
- `*nul_pt` leads to segmentation fault

Pointer and Array

Pointer has its type:

```
int i = 10; int* ipt = &i; ipt++;
```

- pointer ipt move forward 4 byte.

```
char c = 'a'; char* cp = &c; cp++;
```

- pointer cp move forward 1 byte.

Because **integer is 4 byte** and **char is 1 byte**

- pointer always moves multiple of its type

Pointer and Array

array_ptr_1.c

```
1  #define SIZE 5
2
3  int main(void) {
4    int ia[SIZE];
5    char ca[SIZE];
6    int i;
7
8    for(i = 0; i < SIZE; i++)
9      printf("%p\t%p\n",
           &(ia[i]), &(ca[i]));
10
11   return 0;
12 }
```

```
$ ./array_ptr_1
0xbf98b4dc      0xbf98b4f7
0xbf98b4e0      0xbf98b4f8
0xbf98b4e4      0xbf98b4f9
0xbf98b4e8      0xbf98b4fa
0xbf98b4ec      0xbf98b4fb
$
```

For the integer array, the range of address tells you that it occupies a total of **4 x 5 = 20 bytes**.

For the char array, the range of address tells you that it occupies a total of **1 x 5 = 5 bytes**.

Pointer and Array

```
array[i] == *(array + i)
&(array[i]) == array + i
```

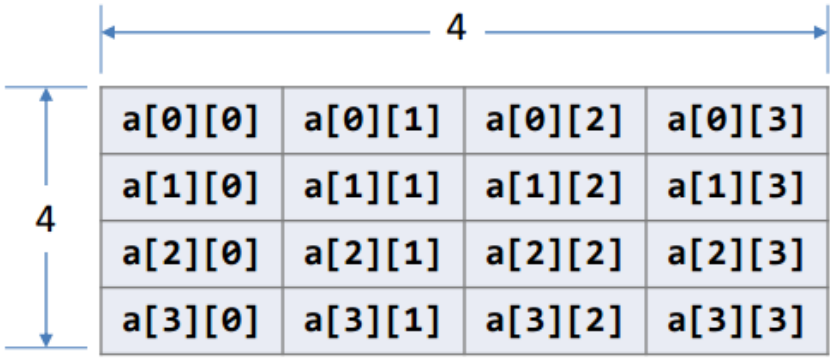
array_ptr_1.c

```
1  #define SIZE 5
2
3  int main(void) {
4    int ia[SIZE];
5    char ca[SIZE];
6    int i;
7
8    for(i = 0; i < SIZE; i++)
9      printf("%p\t%p\n",
           &(ia[i]), &(ca[i]));
10
11   return 0;
12 }
```

array_ptr_2.c

```
1  #define SIZE 5
2
3  int main(void) {
4    int ia[SIZE];
5    char ca[SIZE];
6    int i;
7
8    for(i = 0; i < SIZE; i++)
9      printf("%p\t%p\n",
           ia + i, ca + i);
10
11   return 0;
12 }
```

2D Array



2D_addr.c

```
1  int main(void) {
2    char array[4][4];
3    int i, j;
4    printf("Start = %p\n", array);
5    for(i = 0; i < 4; i++) {
6      for(j = 0; j < 4; j++) {
7        printf("%p ", &array[i][j]);
8      }
9      printf("\n");
10   }
11   return 0;
12 }
```

1. It is a piece of memory with 16 continuous integers.

2. The address '**a**' is a const address and you cannot change it.

3. `a[i][j] == *(a + i*4 + j);`

Match?

Pass by Address

- The caller passes in an address (pass by value)
- The calling function changes the target value by dereferencing the address

pass_by_reference.c

```
1  void basic_math(int x, int y, int *add, int *subtract) {
2    *add = x + y;
3    *subtract = x - y;
4  }
5
6  int main(void) {
7    int add, subtract;
8    basic_math(100, 10, &add, &subtract);
9    printf("Add = %d\n", add);
10   printf("Subtract = %d\n", subtract);
11   return 0;
12 }
```

Pass by Address

The function **indirectly changes** the contents of the target variables through their addresses.

pass_by_reference.c

```
1  void basic_math(int x, int y, int *add, int *subtract) {
2    *add = x + y;
3    *subtract = x - y;
4  }
5
6  int main(void) {
7    int add, subtract;
8    basic_math(100, 10, &add, &subtract);
9    printf("Add = %d\n", add);
10   printf("Subtract = %d\n", subtract);
11   return 0;
12 }
```

Pass by Address

pass_array.c

```
1  void init_array(int array[], int len) {
2      int i;
3      for(i = 0; i < len; i++) {
4          array[i] = i;
5      }
6  }
7
8  void print_array(int array[], int len) {
9      int i;
10     for(i = 0; i < len; i++) {
11         printf("%d: %d\n", i, array[i]);
12     }
13 }
14
15 int main(void) {
16     int a[5] = {0, 0, 0, 0, 0};
17     init_array(a, 5);
18     print_array(a, 5);
19     return 0;
20 }
```

Let us explain how and why after you see the result!

```
$ ./pass_array
0: 0
1: 1
2: 2
3: 3
4: 4
$
```

Memory Allocation

In C, we use `malloc()`

```
void *malloc(size_t size);
```

- `malloc()` allocates a piece of memory of “size” bytes.
- `malloc()` returns the pointer to the created piece of memory.
- **not guaranteed to be all zeros** (use `memset`)

Its opposite function: `free()`

```
void free(void *ptr);
```

- It destroys memory returned from `malloc()`.

Memory Allocation

malloc_1.c

```
1  int main(void) {
2    int *int_ptr;
3
4    int_ptr = (int *) malloc(sizeof(int));
5    if(int_ptr == NULL) {
6      perror("malloc()");
7      exit(1);
8    }
9
10   *int_ptr = 100;
11   printf("%d\n", *int_ptr);
12   free(int_ptr);
13   return 0;
14 }
```

Good practices (1 of 2)

Check the return value of **`malloc()`**. [Lines 4 - 8]

When memory exhausts, **`malloc()`** fails and returns **`NULL`**.

`free()` after **`malloc()`**. [Line 12]

If you don't destroy the memory, it will be destroyed when the program ends.

But, what if your program runs for days? The un-free memory is called **memory leakage**.

Memory Allocation

malloc_1.c

```
1  int main(void) {
2    int *int_ptr;
3
4    int_ptr = (int *) malloc(sizeof(int));
5    if(int_ptr == NULL) {
6      perror("malloc()");
7      exit(1);
8    }
9
10   *int_ptr = 100;
11   printf("%d\n", *int_ptr);
12   free(int_ptr);
13   return 0;
14 }
```

Good practices (2 of 2)

Use **`sizeof()`**. [Line 4]

This can save your life when your code has to run on both 32-bit and 64-bit systems.

Casting **`malloc()`** return value? It is up to you. [Line 4]

http://c-faq.com/malloc/cast.html

Avoid double **`free()`**. [Line 12]

Destroy a memory twice is not fun. Try it out by yourself.

Array Memory Allocation

malloc_2.c

```
1  #define SIZE    5
2
3  void init_array(int array[], int len) {
4      /* same as pass_array.c */
5  }
6
7  void print_array(int array[], int len) {
8      /* same as pass_array.c */
9  }
10
11 int main(void) {
12     int *a = malloc(sizeof(int) * SIZE);
13     if(a == NULL) {
14         perror("malloc()");
15         exit(1);
16     }
17     init_array(a, SIZE);
18     print_array(a, SIZE);
19     free(a);
20     return 0;
21 }
```

Remember, the definition of **`malloc()`**? It gives you a piece of memory of any size.

What is an array? An array is a piece of continuous memory.

So, we can use **`malloc()`** to construct an array!

sizeof ( )

1. `char mystr[100]="test string";`

   `sizeof (mystr) = ?`

   `strlen (mystr) = ?`

2.
```
 5     int days[] = {1,2,3,4,5};
 6     int *ptr = days;
 7     int *ptr1 = (int *) malloc (sizeof (int) * 5);
 8     printf("%u\n", sizeof(days));
 9     printf("%u\n", sizeof(ptr));
10     printf("%u\n", sizeof(ptr1));
```

Memory Utility

`memcpy`: copy memory area

```
void* memcpy(char * dest, const char * src, size_t n)
```

- copy n bytes from src buffer to dest buffer

`memset`: fill memory with constant byte

```
void* memset(char * buf, int constant, size_t n)
```

- set n bytes to the given constant in buffer
- useful when you do some padding

File I/O

file_io_1.c

```
1  int main(int argc, char **argv) {
2    FILE *fp;
3    int c, count = 0;
4
5    *fp = fopen(argv[1], "r");
6    if(fp == NULL) {
7      perror(argv[1]);
8      exit(1);
9    }
10
11   while(1) {
12     c = fgetc(fp);
13     if( c == EOF )
14       break;
15     count++;
16   }
17   fclose(fp);
18   printf("%d bytes read\n", count);
19   return 0;
20 }
```

Opening a file

`fopen()` is the function that opens a file. The return value is a pointer to a type called "**`FILE`**".

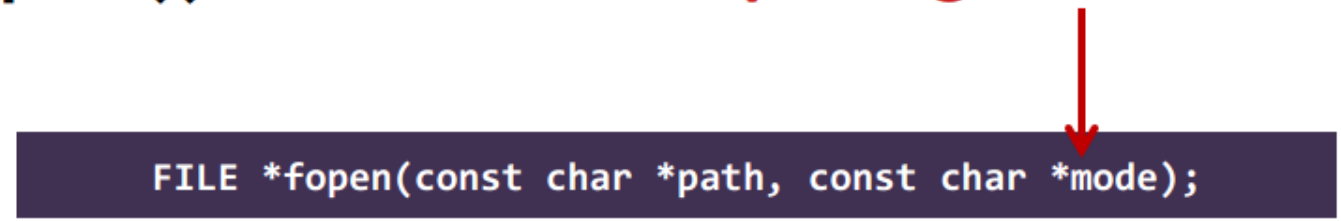
Since you know what a pointer is, this implies that **`fopen()`** allocates memory and returns to you a pointer to a piece of memory of "**`FILE`**" type.

Closing a file

`fclose()` de-allocates every memory associated with the opened file.

File I/O

"**fopen()**" has different **opening modes**.



Common opening mode	Description
"r"	Read only.
"r+"	Read and write.
"w"	Write only. If the target file does not exist, **fopen()** will create such a file.
"w+"	Read and write. If the target file does not exist, **fopen()** will create such a file.

File I/O

file_io_2.c

```
1  int main(int argc, char **argv) {
2      FILE *in_fptr, *out_fptr;
3      int c;
4      if(argc != 3) {
5          fprintf(stderr, "Usage: %s [input] [output]\n", argv[0]);
6          exit(1);
7      }
8      in_fptr = fopen(argv[1], "r");
9      out_fptr = fopen(argv[2], "w");
10
11     while(1) {
12         c = fgetc(in_fptr);
13         if( c == EOF )
14             break;
15         else
16             fputc(c, out_fptr);
17     }
18
19     fclose(in_fptr);
20     fclose(out_fptr);
21     return 0;
22 }
```

“w” - mode

The “**out_fptr**” pointer can only be used for writing.

Try passing it to “**fgetc()**” ☺

File I/O

`fread & fwrite`: binary stream input/output

```
size_t fread(char * buf, size_t unit, size_t num, FILE* stream)
```

```
size_t fwrite(char * buf, size_t unit, size_t num, FILE*
```

- Both function return number of bytes that has been successfully read/written.
- fread reads from **FILE* stream** to **buffer**
- fwrite writes from **buffer** to **FILE* stream**
- **unit** and **num** means read/write **unit*num** bytes

File Utility

`fseek()` – set the file stream position.

```
int fseek(FILE *stream, long offset, int whence);
```

whence	Description
SEEK_SET	Set the position relative to the start of the file. "**fseek(fptr, 0, SEEK_SET)**" goes back to the start of the file.
SEEK_CUR	Set the position relative to the current position. "**fseek(fptr, -1, SEEK_CUR)**" goes back 1 byte. "**fseek(fptr, 1, SEEK_CUR)**" goes forward 1 byte.
SEEK_END	Set the position relative to the end of the file. "**fseek(fptr, 0, SEEK_END)**" goes to the end of the file.

File Utility

```
long ftell(FILE *stream);
```

– report the current stream position in terms of bytes.

```
void rewind(FILE *stream);
```

– It is equivalent to "**`fseek(fptr, 0, SEEK_SET)`**".

```
int feof(FILE *stream);
```

– Report if the stream has reached the end of file.

Reading man pages

Man pages are of the **vital importance** to C programmers.

It includes the following important information:

- Header files (can be more than one) to be included,
- Compiler (gcc) flags to be added,
- Parameter lists,
- Meaning of different return values, and
- Error conditions.

Reading man pages

```
MEMCPY(3)                  Linux Programmer's Manual                 MEMCPY(3)

NAME
       memcpy - copy memory area

SYNOPSIS
       #include <string.h>

       void *memcpy(void *dest, const void *src, size_t n);

DESCRIPTION
       The  memcpy()  function  copies  n bytes from memory area src to memory
       area dest.  The memory areas must not overlap.  Use memmove(3)  if  the
       memory areas do overlap.

RETURN VALUE
       The memcpy() function returns a pointer to dest.

CONFORMING TO
       SVr4, 4.3BSD, C89, C99, POSIX.1-2001.

SEE ALSO
       bcopy(3),  memccpy(3),  memmove(3),  mempcpy(3), strcpy(3), strncpy(3),
       wmemcpy(3)

COLOPHON
       This page is part of release 3.35 of the Linux  man-pages  project.   A
       description  of  the project, and information about reporting bugs, can
       be found at http://man7.org/linux/man-pages/.

                                  2010-11-15                         MEMCPY(3)
```

Reading man pages

Sections of man pages are defined in "**`man man`**".

– E.g., "**`man printf`**" is the same as "**`man 1 printf`**"

- It means the *shell scripting command* "**`printf`**", **not the C library call** "**`printf()`**".
- "**`man 3 printf`**" is to read the manual page of C library call **`printf()`**.

Section	Description
1	Executable programs or shell commands.
2	System calls
3	Library calls
...	...
7	Miscellaneous (including macro packages and conventions)

Simple Debugging with GDB

The most common use of GDB is to track down "segmentation faults"

For best results, always compile your program with "`-O0 -g -Wall`" during debugging

- Three steps to find the origin of segfault
  - `gdb ./a.out`
  - `r`
  - `bt`

Debugging Memory with Valgrind

```
 5  char *chptr;
 6  char *chptr1;
 7  int i;
 8  chptr = (char *) malloc(512);
 9  chptr1 = (char *) malloc (512);
10  for (i = 0; i <= 512; i++ ) {
11    chptr[i] = '?';         // i = 512 invalid W
12    chptr1[i] = chptr[i];  // i = 512 invalid RW
13  }
```

```
  % valgrind ./test
==20410== Invalid write of size 1
==20410==    at 0x40059A: main (test.c:11)
==20410==  Address 0x51f2240 is 0 bytes after a block of size
512 alloc'd
==20410==    at 0x4C2B6CD: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==20410==    by 0x400571: main (test.c:8)
...
```

Recommended Reference

http://c-faq.com/

Spend a few hours to clear up your concept

Thank you